Learning to Program with Haiku

Lesson 11

Written by DarkWyrm

This lesson marks the end of what I would call the "training wheels" of C programming. Once we are done here, we will be getting into what makes C++ powerful and, shortly thereafter, what programming for the Haiku operating system is all about. This lesson's topic is data structures. Even though variables and arrays are great, they are not enough sometimes. Let's get started.

Typedefs and Enumerated Types

It's possible to create your own types in C and C++ using the `typedef` keyword. The `size_t` type that we used in the last lesson with `fread()` and `fwrite()` was created that way. Once we start looking at the specifics of the Haiku API, we'll be using user-defined types all the time. Because the sizes of `long`, `int`, and the other built-in types have sizes that change depending on which operating system is being used, other types have been defined to ensure that a developer can choose a type with a specific size.

A type is defined using the format `typedef baseType newTypeName` like the entries that are found below, taken from Haiku's `<config/types.h>` header.

```
//      The type of data      The new name for the data
typedef signed char           __haiku_std_int8;
typedef unsigned char         __haiku_std_uint8;
typedef signed short          __haiku_std_int16;
typedef unsigned short        __haiku_std_uint16;
typedef signed int            __haiku_std_int32;
typedef unsigned int          __haiku_std_uint32;
typedef signed long long      __haiku_std_int64;
typedef unsigned long long    __haiku_std_uint64;
```

The `signed` and `unsigned` keywords are new only because we haven't needed them yet. Signed types can hold negative values and unsigned ones can't. The highest bit of a signed variable is devoted to indicating a negative value. This affects the range of integer variables, but that's it. For example, `signed char` can hold values from -128 to 127 and `unsigned char` can hold 0 to 255.

These types really are just different names for the same kind of data, so `__haiku_std_uint32` is just another name for an `unsigned int`. The new names created above aren't very friendly, but that's OK. They're not normally used or seen – they are actually intermediary names that are used in some friendlier name definitions found in `<SupportDefs.h>`.

```
typedef     __haiku_int8      int8;       // an 8-bit signed integer
typedef     __haiku_uint8     uint8;      // an 8-bit unsigned integer
typedef     __haiku_int16     int16;      // a 16-bit signed integer
typedef     __haiku_uint16    uint16;     // a 16-bit unsigned integer
typedef     __haiku_int32     int32;      // a 32-bit signed integer
typedef     __haiku_uint32    uint32;     // a 32-bit unsigned integer
typedef     __haiku_int64     int64;      // a 64-bit signed integer
typedef     __haiku_uint64    uint64;     // a 64-bit unsigned integer
```

Ah. Much better. From here on, we'll be using these types instead of the standard ones to get used to seeing them before we dive into the Haiku API.

Enumerated Types

An enumerated type is one which is limited to a specified set of values. For example, if you were making a card game of some kind, you could define a card's suit as an enumerated type that would carry more meaning than an arbitrary integer value. It would look like this:

```
enum card_suit
{
	SUIT_CLUBS,
	SUIT_DIAMONDS,
	SUIT_HEARTS,
	SUIT_SPADES
};
```

This would then make it possible to have a variable `cardSuit` that could have a value that was one of the four in the list. Declaring and initializing it would look like this:

```
enum card_suit cardSuit = SUIT_HEARTS;
```

The format for creating an enumerated type looks like this, with the name and assigning integer values being optional:

```
enum enumName
{
	// A value in an enumerated type can have an integer value assigned to it.
	// Otherwise, the first item in the list will have a value of 0.
	enumValue1 = 5,

	// If a value in the list for an enumerated type is assigned a
	// value, each value after that will be 1 more than the previous one
	// unless otherwise specified. EnumValue2 will have a value of 6.
	enumValue2,

	enumValue3 = 9,

	// enumValue4 will have a value of 10
	enumValue4
};
```

Unions

Unions are a data construct left over from the old days of C when memory was expensive. They're pretty rare anymore except in old code. They are declared like a structure, but it takes up only as much memory as the largest variable inside it. Accessing a member is just like a structure. The key difference between unions and structures is that setting the value on one variable inside a union changes the value for all of the variables inside it. Each variable is a different way of interpreting the value in the memory location that the union occupies. Yeah, it's kind of confusing. Don't worry too much about it.

```
union myUnion
{
	int32 data32;
	int8  data8;
};

myUnion onion;
onion.data32 = 5000;
```

Structures

Structures are groups of variables. They can be of any type and while there is a practical limit to how many different items can be placed in one, there is no technical limit. Structures are used to group together closely-associated data. They are defined like this:

```
// Like enumerated types, the name is optional. Initializing the parts of the
// structure is not possible.
struct myStructName
{
	int8  someInt;
	int16 someOtherInt;
	float someFloat;
	bool  aFlagOfSomeKind;
};
```

Now that we have a structure, we can treat it just like any other data type.

```
myStructName someVar;
myStructName aSecondVar;
```

From here, we access the variables inside the structure using a the structure's name followed by a dot and the name of the variable inside the structure. For example, let's say that we want to set the `someInt` part of `someVar` to 5. It would look like this:

```
// Set the someInt part of someVar to 5.
someVar.someInt = 5;
```

If you want to allocate a structure on the heap, it isn't any different than any other data type.

```
myStructName *ptrStruct = (myStructName*)malloc(sizeof(myStructName));
```

There is a difference, though, in accessing variables inside structures – we use an arrow instead of a dot. The arrow is just a minus sign followed by a greater-than sign.

```
ptrStruct->someInt = 5;
```

Lastly, structures can be nested inside structures. Accessing variables inside them is just a matter of chaining together dots or arrows, as the case may be.

We've seen a lot at once this lesson, so let's take some time to look at some code which puts it to use to get a better grasp of it all. This example is, by far, the longest one we have had. Take the time to slowly go over the code and make sure you understand what each line does before moving on. Go back to previous lessons and look something up if you need to. This is more like "real world" Haiku code than example code from some tutorial. There are some fancy code tricks that we will commonly see in future lessons, but don't worry if you forget them because you'll see them more and more as we go.

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

// A new header! This one is for our rand() function which lets us generate
// sort-of-random numbers
#include <stdlib.h>

// Another new header, but I can't remember what it's for. Hmm....
#include <time.h>

// This Haiku-specific header provides the typedef definitions for uint8,
// uint32, and other types with short, specific names that we saw earlier
#include <SupportDefs.h>

// We'll use an enumerated type to make meaningful values for us to code with.
// We could use #defines, but these are much less likely to cause weird errors.
// Keep in mind that SUIT_HEARTS has a value of 0 when used as an integer and
// SUIT_SPADES has a value of 3. We'll capitalize on their integer values later on
// in this example.
enum card_suit
{
	SUIT_HEARTS,
	SUIT_CLUBS,
	SUIT_DIAMONDS,
	SUIT_SPADES,
	SUIT_NONE		// This is for the jokers, and I don't mean me.
};

// This character array is a lookup table that will make printing the deck MUCH
// easier. Instead of having to putter around with a switch() block, we can
// use the integer value of the items in the card_suit type as an index in this
// list. There's much less typing and it's a little faster than a switch().
static char sSuitCharList[] = { 'h', 'c', 'd', 's', ' ', '\0' };

// This enum holds all of the possible values of the cards. Note that we have
// the integer value of each enumerated value match the card's number, so
// the 10 card has an integer value of 10.
enum card_value
{
	CARD_2 = 2,
	CARD_3,
	CARD_4,
	CARD_5,
	CARD_6,
	CARD_7,
	CARD_8,
	CARD_9,
	CARD_10,
	CARD_JACK,
	CARD_QUEEN,
	CARD_KING,
	CARD_ACE,
	CARD_JOKER
};
```

```
// This is another lookup table. Just like we did with the card suits, we'll use
// the card_value values as integers to look up the string holding the friendly
// name of the card that the user will see. There is one catch: because the first
// card has an integer value of 2, we will have to subtract 2 from the card's
// value to get the proper index in this list.
static char sValueNameList[14][3] = {     "2","3","4","5","6","7","8",
                                          "9","10","J","Q","K","A","Jo"
                                    };


// Using a structure will make it easy to group together a card's value and suit.
struct card
{
      card_value  value;
      card_suit   suit;
};


// This function initializes a standard 54-card deck. Note that it is NOT
// shuffled -- just 2 through Ace in each suit and the two Jokers at the end.
void
InitStandardDeck(card *deck)
{
      // This index variable will be used to keep our place as we work our way
      // through the deck. This is necessary because the index variables in our
      // two for() loops below work their way through the enumerated types for
      // suit and card value.
      uint8 deckIndex = 0;

      // enumerated values can be used just like integers. Instead of just using
      // the letter i like we normally do, we'll use meaningful names for the
      // index variables to make sure that we don't get mixed up
      for (uint8 suitValue = SUIT_HEARTS; suitValue < SUIT_NONE; suitValue++)
      {
            // These two loops walk their way through the deck and assign values
            // to each card in the deck, going in order from 2 through Ace
            // for each suit
            for (uint8 cardValue = CARD_2; cardValue < CARD_JOKER; cardValue++)
            {
                  deck[deckIndex].value = (card_value)cardValue;
                  deck[deckIndex].suit = (card_suit)suitValue;
                  deckIndex++;
            }
      }

      // We have all the number and royalty cards now, so tack on the jokers at
      // the end
      deck[deckIndex].value = CARD_JOKER;
      deck[deckIndex].suit = SUIT_NONE;
      deckIndex++;
      deck[deckIndex].value = CARD_JOKER;
      deck[deckIndex].suit = SUIT_NONE;
}
```

```
// This function takes a pointer, but we're going to use it like an array
void
ShuffleDeck(card *deck, const uint8 &numCards, const uint8 &shuffleCount)
{
	// A deck can have any different number of cards in it. Canasta uses 104,
	// for example. By generalizing the function, we can reuse it for
	// different card games without having to rewrite it. We also have extra
	// flexibility by being able to specify how many times to shuffle the deck
	// to give us extra control over how mixed-up the deck gets.

	// This loop is to shuffle the deck the specified number of times . We can
	// get away with just using i and j for index variable names because we're
	// not using them in the code inside the loops -- they're just for deciding
	// how many times the code inside the loops is repeated and nothing else.
	for (uint8 i = 0; i < shuffleCount; i++)
	{
		// We shuffle the deck by swapping items in the array. It's a little
		// like the ReverseString function from Lesson 7, but we choose random
		// items in the array to swap.

		// The more swaps we do, the better the shuffle, especially with
		// larger decks, so base the number of swaps on the size of the deck.

		// ceil() rounds a floating point number up, regardless of how big
		// the fractional part of the number is. It returns a double, so we
		// will need to typecast it to a uint16 to stop the compiler from
		// complaining. Casting floats and doubles to integers drops the
		// fractional part, but because we've rounded that part off using
		// ceil(), we're not losing anything by doing so.
		uint16 swapCount = uint16(ceil(numCards * 1.25));

		for (uint16 j = 0; j < swapCount; j++)
		{
			// rand() generates a kind-of-random number between 0 and the
			// defined constant RAND_MAX, which is at least 32767. To  make
			// a random number, use the formula
			// randomValue = rand() % rangeOfValues + minimumValue;
			// so making a value from 5 to 12 would be rand() % 7 + 5;

			// Here we randomly select the indexes for two cards in the deck
			uint8 firstIndex = uint8(rand() % numCards);
			uint8 secondIndex = uint8(rand() % numCards);

			if (firstIndex == secondIndex)
			{
				// If the two indexes are the same, we don't want to waste
				// this swap. Because j is incremented every time we go
				// back to the top of the loop, we decrement j to
				// counteract it and get another shot at making a good
				// swap.
				j--;
				continue;
			}

			// Do the card swap
			card tempCard;
			tempCard.value = deck[firstIndex].value;
			tempCard.suit = deck[firstIndex].suit;
```

```
                deck[firstIndex].value = deck[secondIndex].value;
                deck[firstIndex].suit = deck[secondIndex].suit;

                deck[secondIndex].value = tempCard.value;
                deck[secondIndex].suit = tempCard.suit;
            }
        }
}


void
PrintDeck(card *deck, const uint8 &numCards)
{
        // Prints a deck of cards in the format '2h 5c Jc' . They are printed all
        // on one line that may get wrapped around to a second line if the Terminal
        // window is too small to fit it on one.
        for (uint8 i = 0; i < numCards; i++)
        {
                // Here is where we use the card's value and suit to quickly look
                // up the friendly names that the user will see. Check the comments
                // for the definitions for card_value and card_suit near the top of
                // this example for a detailed explanation.
                printf("%s%c ",sValueNameList[deck[i].value - 2],
                                sSuitCharList[deck[i].suit]);
        }
        printf("\n");
}


int
main(void)
{
        // rand() only provides kinda-not-really random numbers. We seed, or
        // initialize, the random number generator with the current time so that it
        // actually provides enough randomness to be useful. We'll do more with
        // time() much later on, so just ignore this for now.
        srand(time(NULL));

        // Our deck of cards
        card deck[54];

        // Initialize our deck of cards
        InitStandardDeck(deck);

        // Display the deck before it gets shuffled
        printf("Our deck of cards before shuffling:\n");
        PrintDeck(deck,54);

        // Shuffle it pretty good -- 5 times should do the trick.
        ShuffleDeck(deck,54,5);

        // Show how mixed-up it is now
        printf("Our deck of cards after shuffling:\n");
        PrintDeck(deck,54);

        return 0;
}
```

Where to Go From Here

Whew! That was a doozy! This example uses a lot of the things that we've spent time learning so far and probably took some time to get through. No bug hunts this time – just some review of the project from the last lesson. Take some time to also go back over the review questions from this unit and the other two units. In the next lesson we will start delving into the part of C++ which gives it *real* power.

Lesson 10 Project Review

In the last lesson, we were presented with the task of making a program that takes at least one filename as a command-line argument and prints it. The steps to do inside our `for()` loop were the following:

1. Try to open the argument for reading as a file.
2. If the open fails, skip to the next iteration.
3. If the open succeeds, try to read a chunk of data from the file, storing the number of bytes read into a variable.
4. Use a `while` loop to read sections of data from the file, repeating while the number of bytes read is greater than 0.
   a. Write the number of bytes read to `stdout`.
   b. Try reading some more data from the file handle, storing the number of bytes read.
5. Close the file's handle.

```
#include <stdio.h>
#include <malloc.h>

int
main(int argc, char **argv)
{
	for (int i = 1; i < argc; i++)
	{
		// open a file handle for reading from argv[i]
		FILE *fileHandle = fopen(argv[i],"r");

		// if the file handle is NULL or there is an error, continue to
		// the next iteration.
		if (!fileHandle || ferror(fileHandle))
			continue;

		// create a data buffer -- an array to hold our data. Size isn't
		// terribly important, but it should be at least a few hundred bytes
		// and no more than about 4000 bytes. You can create it on the stack
		// or use malloc, whichever you prefer.
		char buffer[1024];

		// create a variable to store the number of bytes actually read
		int bytesRead;

		// read data from the file handle and store the number of bytes
		// read into the variable that we just created.
		bytesRead = fread(buffer,sizeof(char),1024,fileHandle);

		// Start our while() loop. Loop while the number of bytes read is
		// greater than zero and if ferror does not indicate an error on
		// the file handle
```

```
		while (bytesRead > 0 && !ferror(fileHandle))
		{

			// write the number of bytes read to stdout
			fwrite(buffer,sizeof(char),bytesRead,stdout);

			// read more data and put the number of bytes actually read
			// into the variable we created above.
			bytesRead = fread(buffer,sizeof(char),1024,fileHandle);
		}


		// free the buffer here if you used malloc, never mind if you put
		// the buffer on the stack.

		// close the file handle here
		fclose(fileHandle);
	}

	return 0;
}
```