

Learning to Program with Haiku

Lesson 1

Written by DarkWyrn



Programming involves communicating your ideas to the computer. The only problem is that computers are stupid. They will do exactly what you tell them and NOT what you mean unless what you mean is, in fact, what you have told them. If you don't completely understand that – believe me – you will soon enough.

The problem with communicating with the computer is that it doesn't speak any human language. All it knows is what to do when given a particular numerical instruction. For people to do this is very tough, so we write in one language accessible to people and the computer turns it into something it can understand. The translation can be done right when the program is run, called an interpreted language, or beforehand as is done in compiled languages. C and C++ are compiled languages, so when writing C++ programs, you write the human language (**source code**) and the compiler turns it into computer instructions (**machine code**).

In C++, all computer instructions are grouped together into blocks called **functions**. Here is an example:

```
void myFunction(void)
{
}
```

Functions may or may not require data to do their job, and they may or may not give back a result when they are done. This one, called `myFunction`, requires no data and returns none. It also does nothing, but that's OK. Here's the format for defining a function:

```
<outdata> <functionname>(indata)
{
    <instructions to do the function>
}
```

The input data is always inside a pair of parentheses and all of the function's instructions will go inside the pair of curly braces. For future reference, all parentheses and curly braces must appear in pairs.

Anyway, the compiler doesn't care how much space is in between all of this stuff, so there is plenty of room to make your code as (un)readable as you like. We could squish all of this together like this:

```
void myFunction(void){}
```

and it would still have the same result. Because whitespace, as it's called, doesn't matter, every instruction (not to be confused with a function) in C++ has a semicolon after it.

For what it's worth, we will use a style of writing code which is quite similar to what the Haiku developers use with a few tweaks. For example, code placed in between a set of curly braces is always indented one level using the tab key.

Here is our first function that really does something:

```
int TwoPlusTwo(void)
{
    return 2 + 2;
}
```

This function, called `TwoPlusTwo`, needs no input data, but returns a result. The result of this function is always an integer – any number without a decimal point. While a number is a number to people, C++ is very particular about keeping track of the types of data passed around.

Every program has one function which needs to be defined: `main`. It can be defined (as in telling the computer what to do for the `main()` function) in a couple of different ways, but we'll use the simplest:

```
int main(void)
{
    return 1;
}
```

This program, when compiled and run, doesn't print anything, but it does return a one back to the OS when it finishes.

Let's actually make this program. Save the above code into a Source Code file named `ReturnOne.cpp`. Open the Terminal and navigate to the folder that has it and type this:

```
gcc -o ReturnOne ReturnOne.cpp
```

This tells `gcc` (GNU Compiler Collection) that you're going to make a program called `ReturnOne` (specified by the `-o ReturnOne`) and you're going to use `ReturnOne.cpp` to make it.

Functions can call other functions, too, but only if the computer knows about them. This will generate an error when compiled:

```
int main(void)
{
    PushTheRedButton();
    return 1;
}
```

Save this as `RedButton.cpp` and compile it with this command:

```
gcc -o RedButton RedButton.cpp
```

The computer doesn't recognize the function `PushTheRedButton()`, so it doesn't know what to do when it sees it and complains. If we tell the computer what it does, it will compile the program properly. Change `RedButton.cpp` to look like this and compile:

```
void PushTheRedButton(void)
{
}

int main(void)
{
    PushTheRedButton();
    return 1;
}
```

PushTheRedButton doesn't do anything, but the computer doesn't care. The other way to call other functions is to use libraries built into the system. We do this by linking a library with our program and telling the compiler what functions are in the library. Now we will do something almost useful. Save the below code into HelloWorld.cpp and compile with `gcc -o HelloWorld HelloWorld.cpp`

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 1;
}
```

There are two parts that are unfamiliar here. The line starting with `#include` tells a part of the compiler called the **preprocessor** to look up function definitions in the system header `stdio.h`. `stdio.h` is the name of a file which defines a lot of standard input and output functions, like `printf()`. The angle brackets – `<` and `>` – tell the preprocessor that these are system headers. We can make our own, but we'll get to that another time.

The other unfamiliar part here is the part inside the parentheses for `printf()`. Everything in the quotes will be printed on the screen if you run this program from the Terminal. This kind of data is called a **string** (as in string of characters).

Question:

What happens if you remove the backslash and the letter `n` from the end of the string so that it looks like this: `"Hello world!"`

Assignment:

Write a program which prints a box on the Terminal using minuses and pipe symbols like this:

```
-----
|           |
|           |
|           |
|           |
-----
```

Question for thought:

How could you write a program which draws two boxes without having to do a ton of typing?

Learning to Program with Haiku

Lesson 2

Written by DarkWyrn



Our first couple of programs weren't all that useful, but as we go along, you'll find that you can do more and more with your programs. This time around we're going to look at two main concepts: comments, and the different stages the compiler goes through to generate your program, and we'll also learn a little about debugging your code.

Comments are notes that you put into the code. They have many uses, such as clarifying a certain section of code, providing warnings, or temporarily disabling a section of code. See the below example for how they can be used.

```
// This is a one-line comment.
// So is this. Everything after the two forward slashes is considered part of it.

int main(void)
{
    PushTheRedButton();           // This code doesn't work.
    return 1;
}
```

Inherited from C is the multiline comment. They start with `/*` and end with `*/` in a way similar to parentheses or curly braces with one difference: you can't put a multiline comment inside another one.

```
/*-----
RedButton.cpp

This code is an example of how the compiler will complain if you use
a function that it doesn't recognize.
-----*/

// This is a one-line comment.
// So is this. Everything after the two forward slashes is considered part of it.

int main(void)
{
    PushTheRedButton();           // This code doesn't work.
    return 1;
}
```

Now we're going to take a step back and look at the process the compiler goes through when building your program. This is important to understand because there are different kinds of errors that can be made when writing code and knowing something about the process will help you find them more easily.

When a program is built from source code, there are four tools that operate on it before it is an executable: the preprocessor, the compiler, the assembler, and the linker.

Stage 1: The Preprocessor

The preprocessor accepts raw source code as input and does a little massaging before the code is sent to the compiler. It removes comments and handles `#include` directives by inserting the contents of the included header file into the code. There are other directives which it handles that will be discussed later.

Stage 2: The Compiler

The compiler translates the C++ code into Assembly language. Assembly is much, much closer to the instructions which the computer understands while still being human-readable. It is also much harder to write programs and is specific to the processor for which it is written.

Stage 3: The Assembler

The assembler creates **object code** from the Assembly code created by the compiler and places them in object files which have a .o extension. Object code is the actual machine-executable instructions used by the computer to run your program. It's not quite ready to run, however. In this state the object files used to make your program are a lot like a set of puzzle pieces that are ready to be put together.

Stage 4: The Linker

The linker pieces together your object files along with any libraries that they use into an executable program.

Debugging

By nature of programmers being human, they make mistakes and lots of them. Writing code and debugging go hand-in-hand and often done at the same time. As such, we will be learning about how to debug programs as we learn about writing them.

Bugs come in two types: syntactic and semantic. Syntactic bugs are easy to find because the compiler finds them for us. These are problems like capitalization errors, missing or extra parentheses, and mistyped function names. Semantic errors are often harder to find because they are errors in the logic of perfectly legal code. A semantic error in English would be “The oxygen sensor on my car needed to be replaced” – the sentence is perfectly legal and correctly constructed, but the word needed is *sensor*, not *sensor*. Examples of semantic errors are extra semicolons in certain places, adding the wrong amount to a number, and making assumptions about the return value of a function.

Let's take a look at a few simplified examples of common errors:

Example 1

Code

```
#include <stdio.h>

int main(void)
{
    return 1;
} }
```

Errors

```
foo.cpp:6: error: expected declaration before '}' token
```

What we have here is an extra curly brace. gcc has given us a fairly cryptic error, as usual, but it has also given us two clues: the file name and the line number. The line number given by gcc and the actual location of the error are not always the same, but in this case they are.

At this point, you might be wondering, “What in the world is a token, genius?” A **token** is any language element. Just as a regular written language has words and punctuation, so do computer languages. Just as two commas in a row are a punctuation error in the English language, having an extra curly brace is a C++ punctuation error.

Example 2

Code

```
/*-----  
RedButton.cpp  
  
/*This code is an example of how the compiler will complain if you use  
a function that it doesn't recognize.*  
  
-----*/  
  
// This is a one-line comment.  
// So is this. Everything after the two forward slashes is considered part of it.  
  
int main(void)  
{  
    PushTheRedButton();           // This code doesn't work.  
    return 1;  
}
```

Errors

```
foo.cpp:7: error: expected unqualified-id before '--' token
```

This is an example of how the line number for the error is not the same place as the actual error. The complaint is about the dashes at the end of the multiline comment at the top. It is caused, however by nesting a multiline comment inside another one. The preprocessor removes all comments, so what the compiler sees is this:

```
-----*/  
  
int main(void)  
{  
    PushTheRedButton();  
    return 1;  
}
```

The compiler doesn't know what to do with the dashed line and complains.

Example 3

Code

```
int Main(void)
{
    return 1;
}
```

Errors

```
/usr/lib/gcc/i486-linux-gnu/4.4.1/../../../../lib/crt1.o: In function `_start':
/build/buildd/eglibc-2.10.1/csu/../sysdeps/i386/elf/start.S:115: undefined
reference to `main'
/tmp/ccv39Cuo.o:(.eh_frame+0x12): undefined reference to `__gxx_personality_v0'
collect2: ld returned 1 exit status
```

This is an error of a different kind. Remember that `main()` needs to be defined in every program? We didn't – we defined `Main()`. The program is otherwise valid, so it compiled just fine, but when the linker attempted to piece it all together, it couldn't find the one required function and threw a hissyfit. Any time you see an error containing `undefined reference to`, you have a linker error.

Resolving `undefined reference` linker errors isn't generally difficult. It usually means one of two things: you forgot to link in a library that you used, or a source code file was accidentally left out when your program was built.

Learning to Program with Haiku

Lesson 3

Written by DarkWyrn



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.

```
#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.

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

```

#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.

```

#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

```

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.

Learning to Program with Haiku

Lesson 4

Written by DarkWyrn



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.

```

#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);
}
```

Learning to Program with Haiku

Lesson 5

Written by DarkWyrn



Computer languages are a funny lot in that some of them are closer to human language and others are closer to machine code. Assembly is one step removed from machine code, yet COBOL is about as close to human language as it gets. C++ is somewhere in the middle, as you will get a hint of in this lesson.

Arrays

In addition to single variables, C++ also gives us the ability to use collections of variables called **arrays**. The individual members of these groups, called **elements**, are all the same type and are clustered together in the computer's memory. You could say that arrays are a little like egg cartons – containers of a bunch of little items of the same kind. Arrays are declared just like regular variables except for a pair of square brackets with a number inside that states how many items the array contains.

```
int thisIsAnArray[5];
```

The above example creates an array containing five integers that take up a contiguous block of computer memory. If you were able to look at a section of memory, the part of the computer's memory which holds our array would look a little like this:

```
[integer0] [integer1] [integer2] [integer3] [integer4]
```

Using arrays can take a little getting used to, however. Each element in an array is assigned a number. Accessing an element in an array is done by using this number inside a pair of square brackets.

```
int main(void)
{
    // Declare the array itself, which contains 5 elements.
    int intArray[5];

    // The first element in an array has an index of 0.
    // More on this in a moment.
    intArray[0] = 10;

    // The second element of the array
    intArray[1] = 11;

    // The rest of the elements in the array
    intArray[2] = 12;
    intArray[3] = 13;
    intArray[4] = 14;

    return 0;
}
```

In this example, we took our 5-element array and set the value of each element to a different number. Notice that the first element in the array has an index – that is, a reference number – of 0. While people start counting items with the number one, computers start counting at zero, so even though the array contains 5 elements, they're numbered 0 through 4. It's this kind of weirdness that programmers deal with that make regular people think programmers are brain damaged or something. Don't worry – it'll get worse.

Arrays require some special care and feeding. The memory used to hold them is set aside in chunks and arrays declared this way have a fixed size. The size of the array doesn't magically increase if we try to set `intArray[6]` to 16. The results are unpredictable, but most often the operating system aborts our program. Using a negative index will also have the same result. This particular kind of error is called a **segmentation fault**, or `segfault` for short. Anyone who remembers the days of Windows 95 probably saw the occasional "General Protection Fault" that commonly appeared. A GPF was just another name for a `segfault`.

Pointers

No discussion of arrays would be complete without also talking about pointers. Pointers are a fundamental concept of C and C++ which can initially sound even crazier than counting from 0. A **pointer** is a variable which contains (or *points to*) a memory address. You can always recognize the declaration of a pointer because its name is preceded by an asterisk (*).

```
char *somePointer;
```

Let's look at a code example which demonstrates how to use pointers.

```
#include <stdio.h>

int main(void)
{
    // This variable will be our "guinea pig."
    int myInt = 5;

    // Declare a pointer that we know doesn't point to a valid memory address.
    int *uselessPointer = NULL;

    // Set the value of this pointer to the memory address that myInt stores
    // its value in. Without the * in front of its name, it would be considered
    // a regular variable and this would generate a compiler error. The
    // ampersand (&) in front of myInt gets the address of myInt.

    // Note that space in between the * and the name of the pointer is OK.
    int *intPointer = &myInt;

    // %p prints the address of a pointer. This changes from one execution of
    // the program to another.

    // A pointer with a * in front of it returns the data its address
    // actually holds, so *intPointer in this case is 5.
    printf("intPointer's address is %p and contains the value %d\n",
           intPointer, *intPointer);
}
```

Pointers, like arrays, require care when used because they can make it really easy for a programmer to `segfault` a program. **Always initialize a pointer to NULL or a known-good address.** What's NULL, you ask? It's just another word for zero when referring to pointers. Even though a NULL pointer is just as unusable as an uninitialized pointer – one which points to a random address – you know for certain that it's unusable.

Strings

In the Lesson 3, we learned about the different kinds of information that can be stored in variables – types – but we left out an important one: strings. We have been using them in `printf()` statements – everything in between a pair of double quotes is a string.

Strings in C and C++ are very different from other data types. A string is an array of `char` values whose last character is a 0. `char` variables can be initialized with either an integer from 0 to 255 or with a character constant – a character enclosed by single quotes – 'a' or 'b'.

In addition to regular letters, there are also some special characters. These special characters all start with a backslash and for the purposes of memory requirements take up one byte even though they take more than one character to type them. Note that these only work with a backslash – a slash which "leans" to the left – and not a forward slash.

| Character | Character Code |
|--------------------------|-----------------|
| Backspace | <code>\b</code> |
| Carriage Return | <code>\r</code> |
| Form feed | <code>\f</code> |
| NULL (string terminator) | <code>\0</code> |
| Newline | <code>\n</code> |
| Tab | <code>\t</code> |
| Backslash | <code>\\</code> |
| Single quote (') | <code>\'</code> |
| Double quote (") | <code>\"</code> |

The first four are not commonly used in Haiku or UNIX/Linux programming. The carriage return (`\r`) is used instead the newline character (`\n`) to start a new line on Macintosh computers. Windows operating systems use both in combination for the same task – `\r\n`. Knowing this is handy when working with text files coming from other operating systems.

Let's look at an example that uses just single characters.

```
#include <stdio.h>

int main(void)
{
    // This loop prints the alphabet in capitals
    for (char i = 65; i < 91; i++)
        printf("%c",i);

    char endl = '\n';
    printf("%c",endl);
}
```

There are tons of different ways to work with strings, so let's look at just a few for the moment. The fastest way to understand it all is with some code. Work slowly through this heavily-commented example to get a good handle on it all.

```
#include <stdio.h>

// A new include! This one has a bunch of functions just for working with strings
#include <string.h>

int main(void)
{
    // Declare a string, aka an array of the char type
    char string[30];

    // Fill the string with 0's. While it might not seem intuitive to include
    // a "memory" function in string.h, it's often used for purposes like this.

    // memset: sets the value of all bytes in a block of memory to a
    // particular value
    // usage: memset(anArray, valueToAssign, sizeOfTheArray);

    // This call sets everything in our array to 0
    memset(string,0,30);

    // Another way to set values of characters in a string: as an array. Here we
    // Individually set the characters. A capital letter A has an integer value
    // of 65.
    for (char i = 0; i < 26; i++)
        string[i] = 65 + i;

    printf("String contains: %s\n",string);

    // Yet *another* way to set a string's value. sprintf() -- think
    // "string printf" -- prints to a string instead of the screen, but
    // otherwise works the same as printf(). Just be careful that what is
    // printed isn't larger than the string that you're printing to. If it is,
    // your program will happily crash into bits.

    // usage: sprintf(aStringVariable,formatString, argumentList)

    sprintf(string,"%f",3.1415927);

    // %s is the placeholder for a string in printf();
    printf("String changed. Now it contains: %s\n",string);

    return 0;
}
```

The reason for using `memset()` in this example needs a little more explanation. Strings, as previously mentioned, are char arrays that are expected to end in a 0 for almost all uses. When we called `memset()`, we set the entire array to zero so that when the first 26 elements is set to capital letters of the alphabet, the 27th element is the terminating null character (0). `sprintf()` automatically places a null terminator at the end. Without this terminator, we end up printing some garbage characters after our string.

Whew! That was a lot of stuff about arrays, pointers, and strings. Lets quickly recap:

- Arrays are declared with a fixed size using square brackets
- Array elements start counting up from 0
- Accessing memory outside the bounds of an array's allocated memory block will cause a segmentation fault (crash).
- An array can be used like a pointer by using the name of the array without the brackets or an index.

- Pointers are variables which hold memory addresses
- Pointers are declared with an asterisk in front: `int *myPointer`
- Pointers should always be initialized either to NULL (zero) or a known-good address
- The address of a variable can be obtained with an ampersand: `&myVariable`

- Character constants are enclosed in single quotes: 'a' or 'X'
- Char variables can be initialized with a character constant or a number from 0 to 255.
- There are special character constants that take more than one character to type, but are treated as one character, such as `\n`, which starts a new line.
- Strings are enclosed in double quotes: "This is a string"
- Strings are arrays of type `char` whose final character is a NULL (0) end-marker

Project

One advantage programmers have over other trades is being able to make tools to help them in their work. Let's make a program which asks for a word from the user and it prints out the integer value of each character.

In order to get information from the user, we'll need to use two new functions: `gets()` and `strlen()`. Both take a `char` pointer as the only parameter. We'll use a `char` array for both – remember that arrays can be used like pointers if you leave out the brackets and index number. Here are what the declarations of these two functions look like and a description of each:

```
char * gets(char *inString);
```

`gets()` gets a string from the user. The user may type as much as he wants and presses the Enter key when finished. The final `\n` character the user types is replaced with a 0 to mark the string's end. `inString` is a `char` array which is to hold the user input. When the user finishes typing, `gets()` copies the user input into `inString` and returns it also. This doesn't sound like it makes much sense, but don't worry about it.

```
int  strlen(char *inString);
```

`strlen()` calculates the length of the NULL-terminated string given to it. A word of warning: passing a NULL string to it will cause your program to crash.

Here are the basic steps for how we will write our program:

1. Make a char array to hold the information from the user
2. Call `gets()` to get the information from the user and store it into our array.
3. Make an int variable and set it to the string's length
4. Use a for loop to print each character in the string both as a character and its numerical value

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

int main(void)
{
    char inString[1024];

    printf("Type the text to convert and press Enter: ");
    gets(inString);

    // Here's where you come in.
    // Use the steps above to figure out what goes here.
    // Steps 1 and 2 have already been done for you.

    return 0;
}
```

Bonus: Make your project print the character code as hexadecimal (base 16) numbers and/or octal (base 8) in addition to regular (base 10) numbers. See Lesson 3 for more information.

Hints: Closely look over the code examples in the earlier section on strings for hints on how to do step 4. Also, have a second look at the list of placeholders used in `printf()` from Lesson 3.

Going Further

Whenever the compiler builds this project, it complains that `gets()` is dangerous and shouldn't be used. Why do you think this might be?

Bug Hunt

Hunt #1

Code

```
#include <stdio.h>

int main(void)
{
    int number = 0;
```

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

Errors

The code builds just fine, but it won't stop running printing stuff on the screen and the only way to stop it is either press Ctrl+C or close the Terminal window.

Hunt #2

Code

```
#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);

    return a + b + c;
}
```

Errors

```
foo.cpp: In function 'int main()':
foo.cpp:12: warning: too few arguments for format
```

Lesson 4 Bug Hunt Answers

1. The `i++` in the for loop needs to be `i += 2`
2. The warnings come from using the `%d` – used for integers – for a float variable. Change the `%d` placeholders in the `printf` statements to `%f` and they'll go away.

Learning to Program with Haiku

Unit 1 Review

Written by DarkWyrn



Let's take a little time to go over what we've learned so far in this unit. We've come quite a distance from the very beginning in Lesson 1 to our first real project in Lesson 5. Try to answer each of the questions below off the top of your head, but don't be afraid to go back and look up the answers if you're stumped or a just a little fuzzy on something. You can find the answers to all of these questions at the end of Lesson 6.

Review Questions

Lesson 1

1. What is machine code?
2. What is source code?

Lesson 2

3. What are the two ways of adding comments to your code?
4. What are the four tools used to turn source code into machine code?
5. Name and define the two general classes of programming errors.

Lesson 3

6. What is the numerical result of $10 \% 4$?
7. What is the difference between `++i` and `i++`?
8. What is a shorter way of writing `myVar = myVar * 5` ?
9. Why is `==` used instead of `=` for comparing two quantities?
10. What is the `printf` placeholder for a float?
11. What are parameters?
12. What is the difference between the types `int` and `long`?
13. What is the numerical result of $10 / 4$? $10.0 / 4.0$?

Lesson 4

11. In the following code snippet, will anything be printed? Why?

```
int i = 0;

if (i)
    printf("Spam and eggs.\n");
```

12. What operator should replace the word OR in the following code: `if (a == b OR a == c)` in order for the `if` statement to evaluate as true if either condition is true?
13. What do each of the three sections of a `for` loop control block do?

Lesson 5

14. What is a segmentation fault?
15. What is the last valid index for an array with 10 elements?
16. Why is it important to initialize a pointer to a known-good value or to NULL?
17. Which of these is a string, `'a'` or `"a"` ?

18. A string is an array of what type?
19. Which header file (`#include`) that we have been using contains helper functions for working with strings?

Learning to Program with Haiku

Lesson 6

Written by DarkWyrn



This time around we are mostly going to expand on things we already know. Back in Lesson 4, we learned about executing pieces of code only if certain conditions are met with `if` statements and we also saw how to repeat a set of instructions with `for` loops. Let's add a few more tools to our kit.

More Loops

Although they are probably the most common, there are more kinds of loops than just `for` loops. In fact, there are two other closely-related loops: the `while` loop, and the `do-while` loop.

`while` loops are simpler, having just a loop condition that causes the code in the loop to be repeated until the loop condition is false. Let's take a look at a way to do our project from Lesson 5 using a `while` loop.

```
#include <stdio.h>

int main(void)
{
    char inString[1024];
    printf("Type some text and hit Enter:\n");
    gets(inString);

    int i = 0;
    while (inString[i])
        printf("String[%d]: %c\n", i, inString[i++]);

    return 0;
}
```

In the above example, we even take a shortcut so that curly braces aren't needed – a postincrement operator increases the index variable `i` after the message is printed. The condition for the loop checks to see if `inString[i]` is a valid character. Integers can be used for boolean logic, that is, true/false tests. Zero is treated as false and everything else is true, so this loop will repeat until the index points to the 0 that terminates the string. Pretty slick, eh?

Aside from the `while` statement and the accompanying condition block, the rest of the loop works just like a `for` loop, so we could just as easily substitute our `printf()` statement with a group of instructions inside a pair of curly braces. One caveat to using this kind of loop: it's pretty easy to end up in an endless loop if you are not careful to ensure that the loop condition is eventually false. In this instance, we would have an endless loop on our hands if the `inString[i++]` were just `inString[i]`.

The `do-while` loop is not used as often as others, but it does come in handy on occasion. It executes the loop instructions once and then checks afterward to see if they need to be done again. Here is the same example using a `do-while` loop.

```

#include <stdio.h>

int main(void)
{
    char inString[1024];
    printf("Type some text and hit Enter:\n");
    gets(inString);

    int i = 0;
    do
    {
        printf("String[%d]: %c\n",i,inString[i]);
    } while (inString[i++]);

    return 0;
}

```

We print a character, check to see if it's zero – which all proper strings end with – and print another character if it's not a zero. Note that a semicolon is required after the `while` condition in this kind of loop unlike the `for` and regular `while` loop. Luckily, if you should happen to forget it, the compiler will complain about it.

```

foo.cpp: In function 'int main()':
foo.cpp:13: error: expected `;' before `}' token

```

There's a problem with using this loop for this job: if the user just hits Enter without typing anything in, it prints a blank character. If the user didn't effectively enter anything, we should skip printing it entirely. This particular task is better suited to using a `for` loop because it requires less work, but we can make using this one work by adding one more bit of code.

```

do
{
    // This will prevent problems caused by the user not typing anything
    if (!inString[i])
        continue;

    printf("String[%d]: %c\n",i,inString[i]);
} while (inString[i++]);

```

The `if` condition here checks to see if the current character is zero – by using a `!`, which evaluates as a boolean NOT. The zero character, which is normally considered false, will make the `if` statement condition true and the program will execute the `continue` statement. `continue` causes program execution to skip directly to the loop's condition block. The zero character that caused the jump to the loop condition will also cause the condition to be false and the loop will exit.

Using `continue` this way, though, is not something you'd normally see. Cases like this normally would call for a `break` statement, which jumps immediately out of the current code block. `break` would skip checking the loop condition and just exit the loop. We'll use `break` statements almost all of the time with our next concept: the `switch` block.

switch blocks

Sometimes we have to deal with making a decision based on one of several possible values for a variable. This could be handled with a series of `if-else` statements, but C and C++ give us `switch` statements to do the job much more elegantly. Let's expand our string-printing project to provide a way to print characters that don't show anything when printed, like spaces and tabs.

```
#include <stdio.h>

int main(void)
{
    char inString[1024];
    printf("Type some text and hit Enter:\n");
    gets(inString);

    int i = 0;
    while (inString[i])
    {
        switch (inString[i])
        {
            case '\n':
            {
                printf("String[%d]: <carriage return>\n",i);
                break;
            }
            case '\t':
            {
                printf("String[%d]: <tab>\n",i);
                break;
            }
            case ' ':
            {
                printf("String[%d]: <space>\n",i);
                break;
            }
            default:
            {
                printf("String[%d]: %c\n",i,inString[i]);
                break;
            }
        }

        i++;
    }

    return 0;
}
```

The value we are evaluating is placed in the parentheses for the `switch` statement. Each value that we want to handle is taken care of by `case` blocks. These `case` blocks define the set of instructions to execute when the value we are evaluating matches the value specified for the `case`. The format for `case` blocks is the following:

```
case valueForCase:
{
    block of instructions
}
```

break statements are very important here because they jump out of the switch statement after handling the particular case. Without one at the end of a case, execution would "fall through" and continue into the next case. This is almost always not what we want. Using our above example, leaving out the break at the end of the case for spaces would cause it to be printed twice: once for the space case and once for the default case, looking something like this:

```
String[5]: <space>
String[5]:
```

The default case is a catch-all for all values which don't fit one of the specified cases. It must be placed at the end of the cases in a switch statement. Any cases placed after the default case will not be executed. It seems silly, but it's true.

Conditional assignment

We've seen some of the ways that C and C++ offer the programmer shortcuts to do certain common tasks, such as adding 1 to a variable. Another such shortcut is the only operator which has three sections. Here's one way to assign a value to a variable based on a condition.

```
int number;

if (someOtherNumber > 5)
    number = 1;
else
    number = 10;
```

Here's another, much shorter, way to do it.

```
int number = (someOtherNumber > 5) ? 1 : 10;
```

Right now, you might be thinking, "*Hold on there, cowboy! What's this here nonsense?*" and needing to lay off the *Bonanza* marathons. Then again, maybe not. The conditional operator has two parts, the question mark and the colon. The format for how it works is as follows:

```
condition ? valueIfConditionIsTrue : valueIfConditionIsFalse
```

Parentheses aren't required around the condition, but some people (like me) prefer to have them even if unnecessary to set the condition apart from the rest. If the condition is true, then the value between the question mark and the colon is returned, otherwise the value after the colon is returned. Its use can be limited, but it does come in handy now and then.

Bug Hunt

Code

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

char *ReverseString(char *string)
{
    // This function rearranges a string so that it is backwards
    // i.e. abcdef -> fedcba

    if (!string)
        return NULL;

    int length = strlen(string);
    int count = length / 2;

    for (int i = 0; i < count; i++)
    {
        char temp = string[length - i];
        string[length - i] = string[i];
        string[i] = temp;
    }

    return string;
}

int main(void)
{
    char inString[1024];

    printf("Type a string to reverse:");
    gets(inString);

    printf("The reversed string is %s\n",ReverseString(inString));

    return 0;
}
```

Errors

The program compiles just fine, but nothing is printed.

Help

Normally no help is given for Bug Hunt sections, but this is harder than many. The bug itself is somewhere in `ReverseString()`. Try using `printf()` calls to print values in certain places to give yourself more information on what the program is doing, like printing the length, the count, etc.

Lesson 5 Bug Hunt Answers

1. The last section of the for loop has just an `i`. It should be `i++`.
2. The variable `c` is missing from the end of the argument list in the `printf` statement.

Lesson 5 Project Review

In the last lesson we were trying to make a program which asks for a word from the user and it prints out the integer value of each character.

We were given these steps to follow to write it:

1. Make a char array to hold the user input.
2. Call `gets()` to get the string from the user.
3. Make an int variable and set it to the string's length
4. Use a for loop to print each character in the string both as a character and its numerical value

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

int main(void)
{
    char inString[1024];

    printf("Type the text to convert and press Enter: ");
    gets(inString);

    // Here was the part we needed to fill in, so here's one way to do it.

    // Step 3.
    int strLength = strlen(inString);

    // Just some fluff to make this program a little nicer
    printf("The character codes for the string '%s'\n",inString);

    // Step 4.
    for (int i = 0; i < strLength; i++)
        printf("[%d]: %c\t%d (0x%x)\n",i,inString[i],inString[i],inString[i]);

    return 0;
}
```

This example even includes printing the character code in both decimal and hexadecimal.

Unit 1 Review Answers

Lesson 1

1. Machine code is the computer's native language – a set of numerical instructions which the computer executes.
2. Source code is the human-readable text used to create a program.

Lesson 2

3. One line comments can be added using two forward slashes – // – and comments which can span multiple lines are placed between these markers: /* */
4. The four tools used to turn source code into machine code are the preprocessor, the compiler, the assembler, and the linker.
5. The two classes of programming errors are syntactic and semantic. Syntactic errors are mistakes in the construction of the computer language, such as having the wrong number of parentheses in a statement. Semantic errors are mistakes in the program logic of code that has correct syntax – bugs in the meaning of the code.

Lesson 3

6. The % operator returns the remainder of the division of the two arguments. $10 \% 4$ is 2.
7. `++i` adds one before the rest of the expression is evaluated. `i++` does the addition afterward.
8. `myVar *= 5`
9. `==` is used for comparison because `=` is used for assigning a value to a variable.
10. `%f` is the placeholder for a float when using `printf`.
11. Parameters are information given to a function which it requires to perform its task.
12. The difference between `int` and `long` is that `long` typically can hold bigger values.
13. $10 / 4$ is integer division, so the result is 2. $10.0 / 4.0$ is floating point division, so the result is 2.5.

Lesson 4

14. Nothing will be printed. `i` has a value of zero, which, for the purposes of logic, is treated as false.
15. The boolean OR operator, `||`, should be used here.
16. The first section of the control block of a `for` loop sets the value of the index variable. The second section is the test to see if another iteration is called for, and the third modifies the index variable.

Lesson 5

17. A segmentation fault is an attempt to access memory that doesn't belong to your program.
18. Because array indices start at 0, the last valid index for a 10-element array is 9.
19. Pointers should be initialized to `NULL` or a known-good address so that you always can tell whether or not a pointer is valid and can, thus, avoid segfaults.
20. `'a'` is a character constant. `"a"` is a string, which amounts to the `'a'` character constant plus a `NULL` string terminator.
21. A string is an array of the type `char`.
22. `string.h` is the header which we have been using that contains helper functions for working with strings.

Learning to Program with Haiku

Lesson 7

Written by DarkWyrn



Some of the most fun times when learning to program are those instances where a neat idea hits you and, after some playing around, you manage to write a program of your own which does something you want and you think "Holy smoke! Check this out! I just made the computer *<insert cool achievement here>*!" This lesson is probably not going to inspire any of those moments, but, like eating your vegetables, it's good for you. Make sure you understand this lesson well before moving on.

Memory: The Heap and The Stack

You are probably aware that every variable takes up some of the computer's working memory. If weren't aware before, you certainly are now. What you may not know is that each program executed by the operating system has two different pools of memory from which variables are created: the heap and the stack.

The **stack** is a fast pool of memory of limited size. When the operating system loads your code into memory before it is run, a section of main memory is set aside for your program's stack memory. Without getting too deep into the details, let's just say that while it is fast, it is also limited and if your program runs out of stack space, it will crash. Using the stack for lots of large arrays is a Bad Idea™. All variables we have used so far were allocated from stack memory.

The **heap** is the main part of the computer's system memory. It takes a little longer to get some memory from it, but you don't have to worry about running out unless you decide to allocate, well, *obscene* amounts of it. A drawback to using heap memory is that you have to take care of making sure that all memory you have received from the system is given back when you're done with it. Not doing so will result in a **memory leak**, which occurs when heap memory is not freed after being allocated. This causes your program to gradually take up more and more system memory until it is finally returned to the system when your program quits, gracefully or otherwise.

Up to now I've mentioned C and C++ in tandem. Everything we've learned so far is exactly the same for both languages. While both methods require pointers – we're dealing with memory addresses here – memory management is one instance where they are quite different. We'll look at the C++ way of doing things when we get to the C++-specific concepts much later on.

The C way of allocating memory is using `malloc()` to get a pointer to some memory for us to use and then calling `free()` on the pointer to return that block of memory to the system. It's also possible to use `realloc()` to change the size of a chunk of RAM originally given to us by `malloc()`. Using these functions properly will take some explanation, so let's have a look at their declarations and handle the details one function at a time.

```
void * malloc(size_t size);
void free(void *pointer);
void * realloc(void *pointer, size_t newSize);
```

Before we even think about what each of these does, it's obvious that there are some types we haven't seen before. The first new type is `size_t`, but it's not really all that new. Instead, it's a new name for one of the integer types that we already know. This kind of trick is often done for **portability** – writing code so that it can be easily written on one operating system and compiled on another. Treat it just like an `int` and everything will be fine.

The other new type we see above is the `void` pointer. `void` pointers don't have a type of their own,

which might sound kind of useless when you consider that we can't use the * operator on them to get their value. That's OK, though. Their usefulness comes from being able to pass around pointers of any type, and when we finally need to get their values, we can convert them to another type by typecasting them.

Typecasting, or just casting for short, is telling the compiler to treat one type as another. The values held in memory don't change, but how they are interpreted does. This will become especially useful later on when we start examining some of the features C++ has over C.

`malloc()` obtains a block of heap memory for us, given a specified size. The address of this block is returned to us as a void pointer, so normally when `malloc()` is called, its return value is typecast to another pointer's type. If `malloc()` can't get a block of the size requested, it returns `NULL`.

`free()` returns heap memory back to the system. When a function like `free` takes a void pointer as a parameter, it simply means that the function will accept a pointer to any data type. No casting is needed to pass, for example, a heap-allocated string to `free()`. Once a pointer is passed to `free()` to return its memory to the system, that pointer it can no longer be used without being reassigned to a valid address. Attempting to use a block of memory that has been freed will have unpredictable results, but normally causes a segfault. If a pointer is intended to be used later on after having its memory freed, it is normally assigned to `NULL` so that it is known to be invalid.

`realloc()` resizes a memory block previously obtained from `malloc()`. It is given the pointer to the block of memory to be resized and the new size desired. If the pointer is `NULL` and the size is greater than 0, then `realloc()` functions just like `malloc()`. If given a valid pointer and a size of 0, it works just like `free`. Both of these cases are not normally used because calling `malloc()` and `free()` directly is much clearer code. `realloc()` may or may not move the memory to a different address and, if it does, it returns the new address. Existing data in the memory block is preserved, even if the pointer is moved to a different address, but any new memory obtained with this call will contain random data.

Working with these memory functions isn't difficult. Let's have a look at some of them in action with a simple example.

```
#include <stdio.h>

// Another new include! malloc.h contains definitions for
// malloc and related memory-allocation routines
#include <malloc.h>

int main(void)
{
    // We are just going to print a string in this example, but
    // we are going to use heap memory this time instead of the stack.

    // This pointer will hold the address of the memory block for our string.
    // We don't have any memory for it just yet, so we'll just set it to NULL
    char *string = NULL;

    // Now we will request a block of heap memory from the operating system
    // and give our string pointer its address.

    // The (char *) is what we need to do to cast our block of memory from
```

```

// having no type (void *) to a char pointer. 50 bytes have been requested,
// which is enough to hold 49 characters and the NULL terminator
string = (char *)malloc(50);

// sprintf is an easy way to convert numbers into strings
sprintf(string, "The number pi is approx. %f", 3.1415927);

printf("%s\n", string);

// This will give the 50 bytes of heap memory that we requested earlier
// back to the system.
free(string);

// The address that our pointer holds is NO LONGER VALID. Any attempt
// to use it will have unpredictable results, but most likely will cause
// a segmentation fault. We're done in this example, but if we were going
// to need the pointer again, we'd need to assign it to a valid address

return 0;
}

```

Beyond Base 10: Binary Math

Now let's take some time to learn some math that we will need in future lessons. Just as computers and people are different in what number they start counting with, the numbering systems that each uses to count are worlds apart.

People use the decimal system, also known as base 10. It is called base 10 because there are 10 possible values in each column, from zero to nine. When you were very young, you may have heard your arithmetic teacher talk about the "ones place," the "tens place," and the "hundreds place." The name for the column is the same as 10 raised to the power of the number of columns moving from right to left, counting from zero.

| "Thousands Place" | "Hundreds Place" | "Tens Place" | "Ones Place" |
|-------------------|------------------|--------------|--------------|
| $1000 = 10^3$ | $100 = 10^2$ | $10 = 10^1$ | $1 = 10^0$ |

For the number 5,234, the total is $(1000 \times 5) + (100 \times 2) + (10 \times 3) + 4$.

The math involved at the bare metal level of interaction with computers is far removed from anything that we, as people, would have any use for. Computers use binary math – base 2 counting. The only values which exist are 1 and 0. The digit "places" in base 2 math look like this:

| "Eights Place" | "Fours Place" | "Twos Place" | "Ones Place" |
|----------------|---------------|--------------|--------------|
| $8 = 2^3$ | $4 = 2^2$ | $2 = 2^1$ | $1 = 2^0$ |

Each individual digit in binary math is called a **bit**. It's just a bit of information, and it doesn't do much on its own. Bits are grouped together into bundles of 8, called **bytes**. Whenever we do any binary math when programming, it will involve at least one byte, and often more, but for now we'll just stick to one to keep things as simple as possible for this confusing topic.

Remember that binary numbers are just a different way of writing numbers, like the difference between the number 68 and the Roman Numeral LXVIII.

To convert a number from binary to decimal, you add up the values in each place. Each digit in a byte is simply a power of two:

| | | | | | | | | |
|----------------------|-----|----|----|----|---|---|---|---|
| Bit Number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Decimal Value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

For each column that has a binary 1 in it, you add the power of 2 that is assigned to that column. For example, binary 10000000 is decimal 128. The only column that has a 1 in it is the first one, which has a decimal value of 128. Binary 10000111 is decimal 135. How? $128 + 4 + 2 + 1$.

It's possible to do addition, subtraction, and any other regular mathematical operation in binary, but it's almost never necessary, so we won't cover it here. There are, however, some other operations that are commonly used which are specific to binary math, and C and C++ provide operators for them. They are remarkably similar to the Boolean logic operators we've already learned.

| Operator | Name | Description |
|-----------------|----------------------------|----------------------------|
| & | Bitwise AND | Turns bits off |
| | Bitwise OR | Turns bits on |
| ^ | Bitwise Exclusive OR (XOR) | Flips bits |
| ~ | Bitwise NOT | Flips all bits in a number |

This table doesn't have the space to give all of the information. Much more explanation is required. These are special mathematical operations with specific purposes. The Boolean AND, OR, and NOT operators are used for program logic, i.e. linking together conditions for `if` statements and the like. The bitwise operators in the table above are for manipulating the bits in numbers.

Bitwise AND

Bitwise AND is used to turn off bits in a number, that is, set certain 1 bits in a number to 0. This is done by comparing the corresponding bits in each number and applying Boolean logic to determine whether or not the bit should be a 1 or 0. Here are two examples that help show how this works.

Example 1: 255 & 240 = 240

| | Decimal | Binary |
|-----|---------|----------|
| | 255 | 11111111 |
| AND | 240 | 11110000 |
| | 240 | 11110000 |

Example 2: 240 & 170 = 150

| | Decimal | Binary |
|-----|---------|----------|
| | 240 | 11110000 |
| AND | 170 | 10101010 |
| | 150 | 10100000 |

The only time a bit stays on is when it is a 1 in the first AND second number. This is why the bitwise AND operator is used for turning bits off.

To turn a specific bit off, we have to figure out what number is needed to turn off only the bit desired and leave the rest on. This is as simple as using a number which has all bits on except for the one we want to turn off.

Let's say that we have a variable which has the value 199 and we want to turn off bit 2 and only bit 2. We'll start with the number 255 – which has all bits on – and subtract 2^2 , which is 2 to the power of the number of the bit we want to turn off. The number we want to use with AND is 251, that is, $255 - 4$. The result of $199 \text{ AND } 251$ is 195.

199 & 251 = 195

| | Decimal | Binary |
|-----|---------|----------|
| | 199 | 11000111 |
| AND | 251 | 11111011 |
| | 195 | 11000011 |

Bitwise OR

Bitwise OR is used in the opposite way to AND – its purpose to turn *on* bits in a number, that is, set certain 0 bits in a number to 1.

Example 1: 192 | 15 = 207

| | Decimal | Binary |
|----|---------|----------|
| | 192 | 11000000 |
| OR | 15 | 00001111 |
| | 207 | 11001111 |

Example 2: 8 | 64 = 72

| | Decimal | Binary |
|----|---------|----------|
| | 8 | 00001000 |
| OR | 64 | 01000000 |
| | 72 | 01001000 |

A bit is turned on whenever either bit is a 1. The math is easier when turning on a specific bit. It is merely a matter of ORing the number we want to change with a value of 2 to the power of the number of the bit. If we have a variable containing 36 and we need to turn on bit 1, then we OR our variable with 2^1 , which is 2.

$$36 \mid 2 = 38$$

| | Decimal | Binary |
|----|---------|----------|
| | 36 | 00100100 |
| OR | 2 | 00000010 |
| | 38 | 00100110 |

Bitwise XOR

Bitwise XOR is probably the weirdest of all of the bitwise operators. XOR stands for eXclusive OR. It is used for inverting bits because it returns 1 if either bit is a 1, but 0 if they have the same value.

$$36 \wedge 255 = 219$$

| | Decimal | Binary |
|-----|---------|----------|
| | 36 | 00100100 |
| XOR | 255 | 11111111 |
| | 219 | 11011011 |

XOR has the occasional useful application, but most of the time, it isn't needed.

Bitwise NOT

Bitwise NOT also flips bits like XOR does, but with less control. It flips all of the bits in a number, just like the example above, but it does not require a second value. Here is an example of how it can be used:

```
int a = 5;
printf("The value of ~%d is %d\n", a, ~a);
```

Shift Operations

In addition to flipping bits on and off, C and C++ provide shift operators which let us quickly do some fast and fancy multiplication and division.

$a \ll b$

Shift the value of a to the left by b places. This multiplies a by 2^b

$a \gg b$

Shift the value of a to the right by b places. This divides a by 2^b

It's a little easier to figure out why it's called shifting by seeing what it actually does to the bits themselves.

| Code | Mathematical Equivalent | Result | Value in Binary (before) | Value in Binary (after) |
|---------|-------------------------|--------|--------------------------|-------------------------|
| 5 << 2 | $5 * 2^2$ | 20 | 00000101 | 00010100 |
| 32 << 1 | $32 * 2^1$ | 64 | 00010000 | 00100000 |
| 64 >> 1 | $64 / 2^1$ | 32 | 01000000 | 00100000 |
| 7 >> 2 | $7 / 2^2$ | 1 | 00000111 | 00000001 |

Knowing shift operations is especially handy when operating with values at the bit level because they allow us to do specific kinds math very, very quickly. Multiplying or dividing by a power of 2 is common in these situations and the equivalent calls to `pow()` or using regular division are far slower.

Bug Hunt

Hunt #1

Code

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

char *ReverseString(char *string)
{
    // This function rearranges a string so that it is backwards
    // i.e. abcdef -> fedcba

    if (!string)
        return NULL;

    int length = strlen(string);
    int count = length / 2;

    for (int i = 0; i < count; i++)
    {
        char temp = string[length - 1 - i];
        string[length - 1 - i] = string[i];
        string[i] = temp;
    }

    return string;
}

int main(void)
{
    char firstString[100],
        secondString[100];
    char *combinedString = NULL;

    printf("Enter your first word: ");
    gets(firstString);

    printf("Enter your second word: ");
    gets(secondString);
}
```

```

    sprintf(combinedString, "%s %s", ReverseString(secondString),
            ReverseString(firstString));

    printf("If you saw these two words in a mirror, it would read '%s'\n",
            combinedString);
}

```

Errors

When run, the program prints "segmentation fault" and nothing else.

Hunt #2

Code

```

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

void MakeBinaryString(char *outString, char valueToConvert)
{
    // Convert a 1-byte value to a string that shows its value in
    // binary. We do this by checking to see each bit is on and
    // if it is, setting the character value to '1' or '0' if not.
    for (int i = 0; i < 8; i++)
    {
        // Is the bit a 1?

        // The shifting is needed to quickly generate a power of 2
        // so that we check only 1 bit at a time, starting with bit 7
        // and working its way down.
        if (valueToConvert & (1 << (7 - i)))
            outString[i] = '1';
        else
            outString[i] = '0';
    }

    outString[8] = '\0';
}

int main(void)
{
    char value = 5;
    char binaryString[6];

    MakeBinaryString(binaryString, value);

    printf("The binary equivalent of %d is %s\n", value, binaryString);

    return 0;
}

```

Errors

When run, the program prints the following:

```
The binary equivalent of 48 is 00000101  
Segmentation fault
```

Lesson 6 Bug Hunt Answers

There was only one hunt because of how hard it was. It's a kind of error called an off-by-1 error. These are sometimes are to track down, like this one was. The errors and corrections are commented.

```
char *ReverseString(char *string)
{
    // This function rearranges a string so that it is backwards
    // i.e. abcdef -> fedcba

    if (!string)
        return NULL;

    int length = strlen(string);
    int count = length / 2;

    for (int i = 0; i < count; i++)
    {
        // This line from Lesson 6 is incorrect
        // char temp = string[length - i];
        char temp = string[length - 1 - i];

        // So is this one
        // string[length - i] = string[i];
        string[length - 1 - i] = string[i];
        string[i] = temp;
    }

    return string;
}
```

The reason why this causes the program to not print anything requires a little thought. The first line in the loop is supposed to save the last character of the string. By not subtracting 1 from the length, it saves the NULL terminator for the string into temp instead. The rest of the loop continues on pretty much as it should. abcdef\0 becomes \0fedcba. Because the NULL terminator is at the very beginning of the string, when printf() goes to print it, it sees the terminator and stops, resulting in the reversed string not getting printed.

Learning to Program with Haiku

Lesson 8

Written by DarkWyrn



Because we've been focusing on the structure of the C++ language so far, we haven't given much attention to getting information in and out of our program. Today we'll change this, but first, let's take a quick look at a few niggling details that we've been able to ignore until now.

Scope

This is not to be confused with mouthwash, though considering the personal hygiene habits of some developers, might not be a bad idea, either. Luckily, on the Internet, nobody knows you're a cat.

Scope in this case applies to variables. A variable's scope is the span of its existence. When a variable is declared, it has a definite beginning, and it can't be used before its declaration in the source code. It also disappears completely when execution reaches the end of the block in which it is declared.

There are three types of scope: local, global, and static. Local variables have the shortest lifespan: they are declared within a code block, such as a function, for loop, or case block, and disappear when execution leaves the block of code.

Global and static variables are declared outside of a function. They can always be accessed, but the order in which they are initialized is not certain, so be careful of globals that depend on the values of other globals. Global variables can be accessed from anywhere. Static variables have the same lifetime as global variables, but they can be accessed only from code within the file in which they are declared. In other words, code in other files can't get at them. All of this can get tricky, but it is easier to understand in code, so let's look at an example.

```
#include <stdio.h>

// This is a global variable, accessible from anywhere
int gAppReturnValue = 0;

int AreaOfSquare(int size)
{
    return size * size;
}

int main(void)
{
    int returnValue = 1;

    // We actually start from 1 in this loop because measuring the area
    // of a square of 0 inches is pretty silly.
    for (int i = 1; i <= 12; i++)
    {
        // i is only accessible from within this loop.

        // If we had a need to utilize returnValue, we could do it here.

        int area = AreaOfSquare(i);
        printf("The area of a square %d inches on a side is "
            "%d square inches\n", i, area);
    }
}
```

```

    // If we try to access the area variable outside the loop,
    // the compiler will complain with an error something like
    // "foo.cpp:30: error: 'area' was not declared in this scope"

    return gAppReturnValue;
}

```

Having seen the above example, you should be aware of what names you give your variables. The lone global variable in this code has a lowercase 'g' for a prefix. This is to signify that it is a global. Static variables use the prefix 's' instead of 'g'. Get into the habit of using these prefixes to avoid more than a few headaches.

If you're like me, you're probably wondering what kind of headaches I'm talking about. Take a look at this example and ask yourself what value would be printed.

```

#include <stdio.h>

// This is a static variable, accessible from anywhere in this file
static int returnValue = 1;

int main(void)
{
    int returnValue = 2;

    printf("The return value of this program is %d\n", returnValue);

    return returnValue;
}

```

In case you're too lazy to try this sample out, the value is 2. This could've been easily avoided by naming our static variable `sReturnValue`. Local variables always receive precedence when there are conflicts in scope. It's also possible to hide a parameter or a previously-declared local variable this way, such as a variable in an `if` block hiding a variable declared at the beginning of a function. *Be careful not to hide parameters with the variables you declare in a function.*

Global and static variables are much like spices in cooking: used sparingly, they can really make writing programs easier, but used too much, all they will do is help you make a big mess. When in doubt, don't use global variables to pass data around – use function parameters instead. Doing so will make it much easier to reuse code that you've written. Once again, work smarter, not harder.

Constants

Not everything is subject to being changed in C and C++. Sometimes we want to ensure that some data not be altered at all. There are other times when we want to use a variable to remember some arbitrary value – we don't care what the value itself happens to be, but we do want to remember its purpose. When we get to writing Haiku programs that use windows and buttons, we will use these quite a lot to specify behavior for controls that we use, such as how they will be resized.

The first kind of constant is the preprocessor definition. In case you don't remember from Lesson 2, the preprocessor is the first tool used when compiling source code into an executable. The preprocessor removes comments, includes header files, and other basic text insertion and substitution. They look like this:

```
#define SOMEDEF "I like cheese!"
#define STRACE(x) printf x
```

The format is simple: `#define <definition> <what gets put there>`. They are simple textual substitutions, much like if you opened a word processor and made it replace all occurrences of `SOMEDEF` with "I like cheese!" As far as the compiler is concerned, you never even typed `SOMEDEF`. You can even make them look like functions, like `STRACE`. We'll look at this one in more detail at the end of the lesson, so just sit tight on that one.

Even more so than arrays, `#defines` need to be handled carefully. It is good practice to put them in all capitals so that they do not look like functions. They also are not type-aware, and one other word of caution: no matter how much they beg, no matter how much they cry, never, **NEVER put a semicolon at the end of a preprocessor definition.**

```
// Do NOT do this
#define THISISBAD 1;
```

Doing so will result in errors in your code that seemingly have nothing to do with the real problem. These are the kind of bugs that make you feel as if you're aging prematurely, even if you're not.

Constant variables are the preferred means of remembering an arbitrary value because they have a defined type. This is done by placing the `const` keyword in the declaration of a variable. Because they are not allowed to be modified, you will almost always see a constant initialized when it is declared.

```
const int someConstantIntValue = 3;
```

Pointers can be made constant, as well, but it gets confusing *very* quickly if you're not careful. The `const` keyword can apply to the pointer's address, to the pointer itself, or both.

```
// This is just a constant integer that we'll use for some of the pointers below.
const int someVariable = 5;
```

```
// These are both pointers to an int where we can change the pointer's address,
// but not its value. These don't have to be initialized.
const int *ptrConstInt;
int const *anotherPtrConstInt;
```

```
// This is a constant pointer. The value itself can change, but we can't
// change the address to which the pointer points. It's pretty useless unless
// we initialize it.
int * const constPtrInt = &foo;
```

```
// These are constant pointers to a constant value. We can't change ANYTHING
// about it, which means that it has to be initialized to be of any use.
const int * const ptrReallyConstInt = &someVariable;
int const * const anotherPtrReallyConstInt = &someVariable;
```

What a mess! There is a rule of thumb which helps make sense of all of this confusion. **The `const` keyword applies to the element to its left. If there isn't anything there, then it applies to whatever is to the right.** In the first two examples above, every time you see `const int` or `int const`, it means that the pointer itself can be changed, but not the value in the address that it points to. Every time you

see `* const`, it makes the pointer's address fixed, but the value at the pointer's address can be changed. The last two examples above combine these two techniques to make everything constant, both address and value. If you're still confused on this, don't worry too much – it's not just you. This is a hard topic.

Using Data From Outside: File Operations

The only way that we know how to get information into our program is `gets()`, and the only way to get it out is `printf()`. While `printf()` is just fine, `gets()` is actually dangerous and whenever the compiler encounters its use, it warns us. The reason is that there is no way to enforce how many characters are placed in the string passed to it. Crashing the program is as easy as typing more characters than the capacity of the array that is given to it. We're going to move on to a much better solution.

Moving information in and out of programs is normally done using streams. Information flows in or out of your program. Direct input from the user is a stream, and the screen is one also – one for output. Console programs utilize streams to get and print information, and they can even be linked together: the program that we use when we run Terminal, called `bash`, features an incredibly powerful ability to take what one program spits out and feed it to another one or dump it into a file.

There are three main streams that are available to every program: `stdin`, the standard input, `stdout`, the standard output, and `stderr`, the error output. Unless changed, a program that gets data from `stdin` will ask for input from the user just like we've done with `gets()` and sending anything to `stdout` or `stderr` will be printed on the screen.

Data can be brought into our programs by reading from a stream, and it can be sent out of our program by writing to one. Some streams are read-only, some are write-only, and some allow both reading and writing. `stdin` is read-only, so we can use it only for getting data. `stdout` and `stderr` are write-only, so we can only print to them. If we create a stream to operate on a file, we can choose either or both.

Each stream has an identifier, called a **handle**. In programming, a handle is just an arbitrary – but most often unique – number which is used to identify an object from others of its kind. Operating on a stream is as simple as obtaining a handle for the stream and calling the appropriate function. Let's take a look at the declarations for some of the functions that we will use in day-to-day C programming.

```
int printf(const char *format, ...);
int fprintf(FILE *streamHandle, const char *format, ...);
```

An old standard. Given a string which defines the format for what is to be printed and an appropriate number of parameters afterward, prints a string to `stdout`. `fprintf()` requires a stream handle to be specified before the format string – making it possible to "print" directly to a file – but otherwise working exactly like `printf()`. When successful, both functions return the number of characters printed. A negative return value is used to indicate failure.

```
int ferror(FILE *streamHandle);
```

This returns a 0 if everything is OK on the stream indicated by `streamHandle` and some other unspecified value if an error has occurred. Make sure that `streamHandle` is not `NULL` – it will segfault your program otherwise.

```
int feof(FILE *streamHandle);
```

This returns a 0 if everything is OK on the stream indicated by `streamHandle` and some other unspecified value if the stream has come to the end of the file. Make sure that `streamHandle` is not NULL – it will segfault your program otherwise.

```
char * fgets(char *array, int arraySize, FILE *streamHandle);
```

`fgets()` is the safe version of `gets()`. It reads in text from the stream handle until it encounters an endline (`'\n'`) character or it reads the number of characters equal to `arraySize`, making a segmentation fault possible if the programmer makes a mistake. As with `gets()`, when successful, `fgets()` returns the same pointer as `array`. If there is an error, it returns a NULL pointer. If `fgets()` comes to the end of the file – only encountered when reading from a file instead of `stdin` – the contents of the array remain unchanged and a NULL pointer is returned. Whenever a NULL pointer is returned by `fgets()`, use `feof()` and `ferror()` to figure out whether you have run out of data or something has gone wrong.

```
FILE * fopen(const char *filePath, const char *mode);
```

`fopen()` opens a file as a stream. If successful, it returns a stream handle which is used by other functions and must be eventually closed with `fclose()`. The mode strings are listed below.

| Mode string | Function |
|-------------|---|
| "r" | Open a file for reading. The file has to exist. |
| "w" | Open a file for writing. If the file exists, its contents are erased and it is treated as a new empty file. |
| "a" | Open a file for writing. Any data written to it is added to the end of the file. If the file doesn't exist, it is created |
| "r+" | Open a file for updating, supporting both reading and writing. The file must already exist. |
| "w+" | Open a file for updating, supporting both reading and writing. If the file exists, its contents are erased and it is treated as a new empty file. |
| "a+" | Open a file for reading and appending. Reading can be done from anywhere in the file, but all writes are tacked onto the end of the file. |

```
int fclose(FILE *streamHandle);
```

Close up an opened stream handle.

Wow. That's a lot of different functions! The scary part is that this is only a very small portion of the functions available. A programmer should always be learning. Once he is comfortable with a language, he is often learning about available functions that are new to him and new ways to use the functions that he knows. The best way to learn how to use functions that are new to you is to use them.

Let's have a look at a program which prints a test file to stdout and creates that file if necessary.

```
#include <stdio.h>

int
FileExists(const char *path)
{
    // This function tests for the existence of a file by trying
    // to open it. There are better ways of doing this, but this
    // will work well enough for our purposes for now.

    // If we were given a NULL pointer, we will return a -1 to
    // indicate an error condition.
    if (!path)
        return -1;

    // Attempt to open the file for reading
    FILE *file = fopen(path, "r");

    // our return value will be 1 if the file exists and 0 if
    // it doesn't. ferror() will return a nonzero result if
    // there was a problem opening the file and a 0 if the file
    // opened OK.
    int returnValue;

    // ferror will crash if given a NULL pointer
    if (!file || ferror(file) != 0)
        returnValue = 0;
    else
    {
        fclose(file);
        returnValue = 1;
    }

    return returnValue;
}

int
MakeTestFile(const char *path)
{
    // Always check for NULL pointers when dealing with strings
    if (!path)
        return -1;

    // Open the file and erase the contents if it already exists.
    FILE *file = fopen(path, "w");

    if (!file || ferror(file))
    {
        // We have a different error code if we couldn't create the
        // file. This makes it possible for us to know if we messed up
        // by passing a NULL pointer or if there was a file-related error.
        fprintf(stderr, "Couldn't create the file %s\n", path);
        return 0;
    }
}
```

```

// The stream handles for stdout, stdin, and stderr are already defined
// for us, so we can use them without any extra work, like in the if()
// condition above and where we put data into our file below.
fprintf(file, "This is a file.\nThis is only a file.\n"
        "Had this been a real emergency, do you think I'd "
        "be around to tell you?\n");

fclose(file);
return 1;
}

int
main(void)
{
    int returnValue = 0;

    // Let's use a test file in /boot/home called MyTestFile.txt.
    const char *filePath = "/boot/home/MyTestFile.txt";

    // Make the test file if it doesn't already exist and bail out of
    // our program entirely if there is a problem creating it.
    if (!FileExists(filePath))
    {
        returnValue = MakeTestFile(filePath);
        if (returnValue != 1)
            return returnValue;
    }

    printf("Printing file %s:\n", filePath);

    // We got this far, so it's safe to print the file
    FILE *file = fopen(filePath, "r");

    if (!file || ferror(file))
    {
        fprintf(stderr, "Couldn't print the file %s\n", filePath);
        return 0;
    }

    char inString[1024];

    // fgets will return a NULL pointer when it reaches the end of the
    // file, so this little loop will print the entire file and quit
    // at its end.
    while (fgets(inString, 1024, file))
        printf(stdout, "%s", inString);

    fclose(file);

    return 0;
}

```

Whew! This is our longest example yet. It's also our closest code to a "real" program. Some idioms, like `if (!file)`, are very common to C and C++ programming, so get used to seeing them. Read over the code in this example and make sure that you understand what each line does.

There are some small changes in the style, as well. Attention to good style is a quirk of the Haiku ecosystem. The Haiku developers, in particular, are notably picky about code which adheres to the OpenTracker code style guidelines and with good reason. Style is partly a matter of opinion, but good code style can also help with debugging and avoiding errors. Bad code style can make it much, much harder. The style we will use throughout the rest of these lessons does not hold to the official Haiku guidelines in certain ways, but it does follow them pretty closely.

Bug Hunt

Hunt #1

Code

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

char *ReverseString(const char *string)
{
    // This function rearranges a string so that it is backwards
    // i.e. abcdef -> fedcba

    if (!string)
        return NULL;

    int length = strlen(string);
    int count = length / 2;

    for (int i = 0; i < count; i++)
    {
        char temp = string[length - i];
        string[length - i] = string[i];
        string[i] = temp;
    }

    return string;
}

int main(void)
{
    char inString[1024];

    printf("Type a string to reverse:");
    gets(inString);

    printf("The reversed string is %s\n",ReverseString(inString));

    return 0;
}
```

Errors

```
foo.cpp: In function 'char* ReverseString(const char*)':  
foo.cpp:18: error: assignment of read-only location '* (string + ((unsigned int) ((length + -0x000000000000000001) - i)))'  
foo.cpp:19: error: assignment of read-only location '* (string + ((unsigned int)i))'  
foo.cpp:22: error: invalid conversion from 'const char*' to 'char*'
```

Answers from Lesson 7's Bug Hunt

1. The pointer combinedString doesn't point to a valid memory address. It needs to either be given heap memory by malloc() – which is later freed – or declared on the stack as an array.
2. The size of binaryString array in main() is too small. It needs to be at least big enough to hold 1 character per bit in a byte plus 1 for the NULL terminator, so binaryString must be at least 9 characters instead of 6.

Learning to Program with Haiku

Lesson 9

Written by DarkWyrn



If you haven't noticed, we've come a long way since Lesson 1. We're steadily getting closer to being done with learning the basics of C and C++. This time around, we'll expand what we know about arrays and pointers.

Arrays: Pointers with Smoke and Mirrors

When we first looked at strings in Lesson 5, we saw how arrays and pointers are pretty closely related, but we didn't look close enough to discover how close they really are. Arrays are really just a friendly way of using pointers. Let's look at two different ways of changing the same string that demonstrate this.

```
#include <stdio.h>

int
main(void)
{
    char string[30];

    // Set the array to the lowercase alphabet using the way we already
    // know: brackets

    for (int i = 0; i < 26; i++)
    {
        // Character constants can be treated like integers. It saves us from
        // having to look up to see what integer value the lowercase letter
        // A has.
        string[i] = 'a' + i;
    }
    string[26] = 0;
    printf("%s\n", string);

    // Here's another way: using pointers and some math
    for (int i = 0; i < 26; i++)
    {
        char *index = string + (i * sizeof(char));

        // Use a capital A just for something different
        *index = 'A' + i;
    }
    string[26] = 0;
    printf("%s\n", string);
}
```

Both loops do the exact same thing except for using capital letters in the second one. The code for the second one looks really strange, though, so let's pick it apart, starting with `sizeof()`. `sizeof()` is a language feature which works like a function. It can be given a variable or the name of a type and it will return the size, in bytes, that one object of that type takes up in memory. For example, `char` objects are only 1 byte large, but an `int` takes of 4 bytes. Give `sizeof()` an array and it will return the amount of memory the entire array occupies. Because the size of a `char` is 1, we could have skipped the multiplication entirely, but it's a good idea to get into the habit of avoiding assumptions like this. You'll have a fewer gray hairs that way.

The next strange trick we see in the code above is adding a number to a pointer. Pointers are just variables that contain memory addresses, so adding an integer to a pointer just changes the memory

address. We can't change the array's address, so we've created a pointer that we can change. This allows us to add the size of an array element to it. As we do this over the course of the loop, the pointer will hold the address of each character in the array.

| Array Index | Equivalent Pointer Math | Value |
|-------------|-------------------------|-------|
| string[0] | index | a |
| string[1] | index + (size * 1) | b |
| string[2] | index + (size * 2) | c |
| string[3] | index + (size * 3) | d |

We can use this close link between arrays and pointers to pull off other fancy tricks, like initializing an entire `bool` array to true. How? `memset()`, which we haven't used in quite some time.

```
int
MyFunc(void)
{
    bool bArray[100];
    memset(bArray, 1, sizeof(bool) * 100);
    return 0;
}
```

This little function just sets each byte in the array to a 1, which is true for Boolean logic. Not only is it less work, it is much, much faster than using a loop to set each element individually.

Multidimensional Arrays

It's really inconvenient that we can make arrays of any kind of data except strings. Strings are everywhere in everyday programming, but we can't make an array of arrays, can we? Actually, yes we can. To have arrays of strings, we must conquer one of the most potentially confusing parts of C and C++: multidimensional arrays. Not to worry, I'm going to make this as simple as possible.

Regular arrays are best thought of as just a series of elements in a row:

```
int myArray[10];
```

myArray:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Arrays can have more than one dimension – for example, a two dimensional array can be thought of as a grid – a group of rows. This one has two rows with five elements in each row.

```
int my2DArray[2][5];
```

my2DArray:

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |

The easiest way to think of an array with more than one dimension is to think in terms of spatial dimensions, working from right to left as we add another pair of brackets of each dimension. One dimension is a line, two is a rectangle, and three is a cube. Beyond three, it's better to think of the dimensions as groups of cubes or groups of cube groups. Declaring and accessing an array with more

than one dimension is just a matter of adding another set of brackets when declaring and accessing the array.

| Number of Dimensions | Array Declaration |
|----------------------|--|
| 1 | my1DArray [elementCount]; |
| 2 | my2DArray [numRows] [itemsInRow]; |
| 3 | my3DArray [numGrids] [rowsInGrid] [itemsInRow]; |
| 4 | my4DArray [numCubes] [numGrids] [numRows] [itemsInRow]; |
| 5 | my5DArray [numCubeGroups] [cubesInGroup] [gridsInCube] [rowsInGrid] [elementsInRow]; |

Now that we have a handle on how multidimensional arrays are declared, let's put them to use in some sample code.

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

int
main(void)
{
    // Declare and initialize an array with 4 rows of 5 items each – a
    // grid 5 elements wide and 4 elements high
    int integerArray[4][5];

    int value = 0;
    for (int y = 0; y < 4; y++)
    {
        for (int x = 0; x < 5; x++)
            integerArray[y][x] = value++;
    }
    return 0;
}
```

This code snippet declares an array and initializes it with a loop. Because the memory for the entire array is allocated in one large block, we can use `memset()` in combination with `sizeof()` to set every byte in the array to the same value. This also means that we can use a pointer to the same address as our two dimensional array to treat the whole thing as one long list.

```
int
main(void)
{
    // Declare and initialize a grid of integers which has 4 rows of 10
    // integers each.
    int intArray[4][10];

    for (int y = 0; y < 4; y++)
    {
        for (int x = 0; x < 10; x++)
            intArray[y][x] = (y * 10) + x;
    }

    // Even though it was declared as a grid, sometimes it's just a lot easier
```

```

// to think in terms of one long list of 40 elements. This is just a
// different way of looking at the same set of data. Because this is a
// two-dimensional array, intArray by itself is of type int ** -- a
// pointer to an integer pointer. Adding an asterisk makes it just an int *.
int *pInt = *intArray;
for (int i = 0; i < 40; i++)
    printf("%d\n", pInt[i]);
}

```

Even though we have initialized all of our arrays using `memset()` or a loop, it's possible – and sometimes necessary – to use many different arbitrary values. This is done using a comma-separated list of values inside a pair of curly braces. A pair of braces is needed for each dimension. Here is the same basic code changed to initialize the array without using a loop.

```

int
main(void)
{
    // Declare and initialize 4 integer arrays which have 10 elements each.
    int intArray[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                           { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 },
                           { 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 },
                           { 30, 31, 32, 33, 34, 35, 36, 37, 38, 39 } };

    int *pInt = *intArray;
    for (int i = 0; i < 40; i++)
        printf("%d\n", pInt[i]);
}

```

It's a lot more typing, but if we had a list of different values that didn't follow any pattern at all, this would be our only option. One drawback to this method is that we have to set the values for all of the elements – we can't just pick and choose which ones we want to set. It also gets increasingly difficult to read for arrays having more than two dimensions. If this were just a plain array, we'd just need one set of curly braces and a list of values in it, like this:

```
float someArray[3] = { 1.1, 2.2, 3.3 };
```

This is also one of the few instances where you have to have a semicolon outside a pair of curly braces.

To create a list of strings, we simply create a two dimensional char array. Although we could use a series of comma-separated character constants, C and C++ give us a couple of kinds of shortcuts to save a lot of typing and possibly some counting when initializing strings.

```

// This is the hard way. What a mess!
char myShortString[15] = { 'a', 'b', 'c', 'd', 'e', '\0' };

// Declare an array that can hold up to a 15 character string, including
// NULL terminator. This is MUCH better than using character constants and braces.
char myFastString[15] = "abcde";

// Leaving out the size tells the compiler to allocate enough memory to hold
// the string. This saves calling strlen() or counting it ourselves.
char myLongString[] = "This is some really long string I don't have to count.";

// We can leave out the size on one-dimensional arrays of other types if we

```

```
// initialize them.  
int myIntArray[] = { 0, 1, 2, 3, 4, 5 };  
  
// This has the same result as the declaration for myLongString. It is also  
// the more common way of doing it.  
char *anotherLongString = "This is some other really long string.";
```

One other useful tip: **don't** try leaving out the size on arrays with more than one dimension. It can only be done with the leftmost dimension listed when declaring a multidimensional array and mixing the two can only lead to confusion. Trust me on this one.

Once again, we've covered a lot in a short amount of time, so let's quickly review:

- Integers can be added to or subtracted from the address held by a pointer.
- `sizeof()` returns the size of a type, variable, or array, measured in bytes.
- You can use pointer math to get the value of an element in an array.
- The memory allocated for an array is contiguous.
- Arrays can be declared to have more than one dimension and accessed that way or reinterpreted as one long list of elements using a pointer.
- `char` arrays (strings) can be initialized using a regular string.
- Non-string arrays are initialized using curly braces.
- All elements in an array must be given a value when it is initialized.
- You can leave out the size in the brackets on one-dimensional initialized arrays if you just want enough memory reserved to hold the values given to it.

Answers from Lesson 8's Bug Hunt

1. The problem is that the parameter `string` is a `const char *` instead of a just `char *`. This means that the string given to `ReverseString()` can't be changed and the compiler complains when we try to change it in the `for` loop. Remove the `const` keyword and everything is happy.

Learning to Program with Haiku

Unit 2 Review

Written by DarkWyrn



We've only covered four lessons, but they sure were packed with information. Let's make sure you haven't forgotten much, if anything. You can find the answers to all of these questions at the end of Lesson 10.

Review Questions

Lesson 6

1. What are the three types of loops?
2. When is a `switch` statement better suited to making a decision than an `if` statement?
3. What happens when the default case in a `switch` statement is not the last case in the list?
4. Rewrite the following code to use the conditional assignment operator to set the value of `myVariable`.

```
if (myInt > 5)
    myVariable = 50;
else
    myVariable = 100;
```

Lesson 7

1. Which memory pool is bigger, the stack or the heap?
2. What is a memory leak?
3. What is portability?
4. What is it called when you tell the compiler to treat one type as another?
5. What is the difference between `malloc()` and `calloc()`?
6. What is the numerical result of `7 & 0`?
7. What is the numerical result of `7 | 8`?
8. What does 33 look like in binary?
9. What is the binary number 1000010 in regular decimal notation?
10. What is the numerical result of `1 << 5`?

Lesson 8

1. What is the scope of a variable?
2. What are the three types of scope?
3. What part(s) of the following pointer declaration is constant: the value, the pointer's address, or both?

```
int * const myPointer;
```

4. Where does anything written to `stderr` normally go?
5. What is a handle?
6. If you want to write data to the end of an existing file, what mode must be specified in a call to `fopen()`?

Lesson 9

1. How many rows are in the following multidimensional array: `float myFloatArray[10][20];`

Learning to Program with Haiku

Lesson 10

Written by DarkWyrn



Back when we first took a look at pointers, I mentioned that they are a powerful tool. Today we'll see how they can be used with functions and learn some different ways of passing variables to functions.

Default Values for Parameters

When calling functions, sometimes there are certain parameters that almost never change. Let's say that in a program we're writing we have a function to create a window on the screen that looks like this:

```
void MakeWindow(int left, int top, int right, int bottom, const char *title,
                int type, int look);
```

Each time we use this function, we end up having different values for the location, size, and title, but the type and the look don't change – almost all of the time the same numbers for type and look are used because we're making regular windows. This makes for extra typing, so we may want to use a default value for the type. To do this, we change the declaration slightly:

```
void MakeWindow(int left, int top, int right, int bottom, const char *title,
                int type = REGULAR_WINDOW_TYPE, int look = REGULAR_WINDOW_LOOK);
```

We are, in a sense, initializing the parameter so that we don't always have to type in a value when we call the function. Parameters with default values always go at the end of a parameter list in a function call. In this particular case, a lot less typing is involved for the usual case of a “regular” kind of window:

```
MakeWindow(100, 100, 500, 400, "MyWindow", REGULAR_WINDOW_TYPE, REGULAR_WINDOW_LOOK);
```

becomes

```
MakeWindow(100, 100, 500, 400, "MyWindow");
```

We won't see too many instances of default parameter values until we start writing programs that use Haiku's graphical toolkit, but it's good to know about these beforehand.

References

References, in this case, are not something you would find in the library. Instead, they are a hybrid of a pointer and a regular variable. They are treated like a regular variable in terms of not having to use the an asterisk to get its value, but they don't have their own memory for storing data. Instead, they use another variable's space. Here is an example of a reference declaration:

```
int myInt;
int &myRef = myInt;
```

The ampersand (&) in between `int` and `myRef` is the difference between declaring a regular variable and a reference. In this example, `myRef` is a reference to `myInt`. `myRef` is, for all practical purposes, just another name for `myInt`. Changing one will change the value of the other. This works with pointers that have a valid memory location, too. Observe:

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

```

int
main(void)
{
    // Create a pointer with some heap memory and initialize it
    int *myPointer = (int*) malloc(sizeof(int));
    *myPointer = 1;

    // Create a reference to the myPointer's location.
    int &myRef = *myPointer;

    myRef++;

    printf("The value at myPointer's location is %d\n", *myPointer);

    free(myPointer);

    return 0;
}

```

References are a feature that makes you want to say, "OK, that's nice. Let's move along now," but there's more than meets the eye here. First of all, you can't change the location that a reference points to, so `myRef` will always refer to the address that `myPointer` points to – even if `myPointer` changes – so trying to use `myRef` after `myPointer` has been freed will cause a segmentation fault. This unchangeability is actually a good thing. A reference has to be initialized and can't be uninitialized, so it will always be valid unless the memory it refers to has been freed or if the variable it refers to goes out of scope. This makes references a great deal safer than pointers at the expense of some flexibility.

Parameter Use: by Reference or by Value

There is more than one way to get data out of a function. The usual way is the value it returns, but references provide another. Normally, when a parameter is passed to a function, a copy of the variable is passed to the function – the *value* of the parameter has been sent to the function, not the variable itself. This means that you can change the parameter without causing trouble outside the function. When a reference to a parameter is given to a function, any changes that the function makes to it continue on after it exits. Changing a function to pass a parameter by reference is as simple as adding an ampersand (&) in front of the parameter's name.

```

const char * someFunction(int &integerByReference);
float someOtherFunction(const double &doubleByReference);

```

In this example, not only does `someFunction` spit out a string, but it can also tweak `integerByReference`, allowing us to get two values from one function call instead of just one. `someOtherFunction` uses a constant reference. While this might seem silly – passing a reference that you can't change – but it does have a purpose: it saves a copy. This is one way to speed up a function that is frequently called, especially if it has a lot of parameters or if a parameter takes up a lot of memory.

Just to make sure that we have a pretty good handle on the difference between passing a variable by value and passing one by reference, let's look at a code example that illustrates the difference.

```

#include <stdio.h>

// This function passes x by reference and y by value
int
myFunction(int &x, int y)
{
    x = x * 2;
    y = y + 5;
    return x * y;
}

int
main(void)
{
    // Create a couple of variables for our work
    int foo = 5;
    int bar = 10;

    int outValue = myFunction(foo,bar);

    printf("foo is %d, bar is %d, and myFunction(foo,bar) is %d\n",
           foo,bar,outValue);

    return 0;
}

```

In this example `foo` starts with a value of 5, but because `myFunction()` changes it, its value is 10 when it is printed. `bar` is not changed because `myFunction()` changes a *copy* of `bar`.

Pointers to Functions

Just when you thought pointers couldn't get any weirder, it gets worse. It is possible to not only have a pointer whose address holds a value, pointers can point to an address containing code.

```
void (*functionPointer)(int value, int anotherValue);
```

This snippet of code is **not** a function. It is actually a declaration of a pointer called `functionPointer`. Parentheses are placed around the asterisk and name of the pointer to make sure that it is declared as a function pointer instead of a function which returns a void pointer.

Types for function pointers are very specific. The return value and the number and types of parameters are all part of a function pointer's type. These two function pointers are not the same type:

```
void (*integerFunction)(int value);
int (*anotherIntegerFunction)(int value);
```

Executing a function by way of a pointer is dead easy: Treat the pointer as the name of the function. This example calls the function held by the `integerFunction` pointer:

```
integerFunction(5);
```

Like references, the uses of function pointers are not immediately obvious. They add incredible flexibility to a program. Code can be bolted on or changed out just like parts on a car. Interpreted

languages, such as Python or Perl, make it very easy to change a program on the fly, but this is not very easy at all for a compiled language like C++. Although normally pretty rare, we'll use function pointers quite a bit when we look at program addons later on. For now, don't worry too much about them.

Pointers to pointers

Yes, pointers can point to pointers. The address a pointer holds can easily be the address of another pointer. This is just a matter of adding a second asterisk when declaring a pointer.

```
char **somePointerToAPointer;
```

Don't forget that a pointer's declaration doesn't allocate any memory! The only thing that exists after this declaration is `somePointerToAPointer`. We can use it for different things, such as getting a pointer from a function without using a return value or creating a list of strings. Yes, this is the way you create a fixed list of strings on the heap. This is also how we get arguments from the command line.

Command Line Arguments

Just like functions taking parameters – or arguments... they're the same thing – programs themselves can have information passed to them. Take, for example, this Terminal command:

```
$ rm -f --verbose myFile
```

The command `rm` has three arguments: a filename and two switches. Command line switches are options that change the behavior of a program without being the information on which it operates. In this case, `rm` operates on the file `myFile`. The `-f` switch tells it to force removal, not asking for confirmation, and the `--verbose` switch tells it to print more information to the screen than it normally does.

Switches in Windows begin with a slash, but for Linux, OS X, and Haiku, they begin with a dash. As a general rule, switches with only one dash have only one letter and switches with two dashes are generally words and phrases separated by a dash. We're not going to focus too much on command line switches for the purposes of these lessons just because our projects will not be complex enough to warrant using them.

In order for a program to take advantage of command line arguments, the way we use `main()` must change a little bit:

```
int
main(int argc, char **argv)
{
    return 0;
}
```

Now `main()` takes two parameters: `argc`, which is the number of arguments from the command line and `argv`, which is a list of strings which contain the command line arguments. Treat `argv` like an array – if `argc` is 2, then `argv` will have elements numbered 0 and 1. `argv[0]` always contains the name of the program when it was run. This program will print the command line arguments passed to it.

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

```

int
main(int argc, char **argv)
{
    // Iterate through all arguments and print them.
    for (int i = 0; i < argc; i++)
        printf("Program argument %d: %s\n", i, argv[i]);

    return 0;
}

```

The place to look closely is the end of the `printf()` statement. Using a pointer to a pointer is literally just like using a multidimensional array. Strings are just special `char` arrays, so we are accessing lists of characters individually here. If we wanted to use just the second character of the first argument, we'd use `argv[0][1]`. In case it's a little fuzzy, bracket order goes from the largest group to the smallest from left to right, so we would be accessing element 0 in the list of lists and choosing element 1 – the second character – from that list.

Project

With everything we've learned in these ten lessons thus far, we have the capacity to do a great deal. This will be our first project of notable value. We are going to write a simple version of the command line utility `cat`, which concatenates, or joins together, files by printing them to `stdout` in the order that they are specified from the command line.

For this project, we will need to use two new functions: `fread` and `fwrite`.

```

size_t fread (void *buffer, size_t size, size_t count, FILE *stream);
size_t fwrite (void *buffer, size_t size, size_t count, FILE *stream);

```

`fread` reads in data from a file stream. This function will try to read in `size * count` bytes and place the data into `buffer`. This is really handy because you can allocate any kind of array for `buffer`, use the `sizeof()` function on its type for `size`, and send the number of elements in the array as `count`. `fread` returns the number of elements actually read. If it is not the same as what was requested, there was either an error or the end of the file was reached.

`fwrite` works the same way as `fread`, but in reverse. Data in `buffer` is written to `stream` and the number of elements written are returned.

Conceptually, this will involve what we have just learned about using command line arguments in combination with file operations from Lesson 8 plus `fread` and `fwrite`. Use a `for` loop to execute the following set of steps on each argument.

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

Hints, Warnings, and Advice

- Printing file errors using `ferror` might be a nice touch.
- The chunk of memory you store the file data in can be from the stack or from the heap.

Because we haven't done much actual writing of code, I'll do a little to get you going. All you'll have to do is replace each of the comments with the corresponding code. It might be a good idea to write a little bit at a time and recompile. Writing code incrementally and testing it helps keep bugs small and easy to locate.

```
#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]

        // if the file handle is NULL or there is an error, continue to
        // the next iteration.

        // 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.

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

        // read data from the file handle and store the number of bytes
        // read into the variable that we just created.

        // 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
        {
            // write the number of bytes read to stdout

            // read more data and put the number of bytes actually read
            // into the variable we created above.
        }

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

        // close the file handle here
    }

    return 0;
}
```

Learning to Program with Haiku

Lesson 11

Written by DarkWym



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.

```

#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;
}
```

Learning to Program with Haiku

Lesson 12

Written by DarkWyrn



C++ is the language in which most Haiku programs are written. It is an extension of the C language written by Dennis Ritchie at Bell labs in the 1970s. The language fundamentals that we have been learning are common to both languages, but from here on out we will be learning the parts of C++ that C cannot do. It is these parts that make it powerful, but to really grasp them well, we must change our perspective of writing code.

Object-Oriented Programming

The programs that we have been writing have been part of a paradigm called procedural programming, which is development that centers around individual function calls. This is fine for many purposes, but programming for the Haiku Graphical User Interface (GUI) goes beyond this to another paradigm called object-oriented programming.

The idea behind object-oriented programming is that an application consists of the interaction of objects to perform a task. Each object is a "black box" which has certain properties and ways of interacting with it while hiding how it actually works. All we know about the object is how to interact with it. A real life example would be a television: pushing buttons changes the channel, adjusts settings in the on-screen menu, adjusts the volume, and turns it on or off. We also know its size, color, and approximate weight, and unless you're a TV repairman, you probably don't know how it works or have access to its internal components.

A C++ object is very much like a television. It has functions, called **methods**, which provide a means to interact with it. It also has variables for holding data which are called **properties**. Not all of an object's methods and properties are necessarily accessible by outsiders.

Classes

Type definitions in C++ are called **classes**. Classes are defined with code very much like the data structures we learned about in the last lesson. Here is an example class definition for our television.

```
class Television
{
public:    // All of these functions are accessible by anyone. Without the
        // 'public' keyword, all of these would be strictly internal methods -
        // the default access mode is private.

        // This is the constructor function. More on this in a moment.
        Television(void);

        // This is the destructor function. More on this in a moment,
        // too.
        ~Television(void);

    void    SetChannel(uint16 channel);
    uint16  GetChannel(void);

    void    SetVolume(uint8 volume);
    uint8   GetVolume(void);

    void    TogglePower(void);
    bool    IsTurnedOn(void);
```

```

private:    // Everything below this word can't be touched by the outside world

    int8      DetectActiveInput(void);
    void      ConnectToSignal(void);

    uint16    fChannel;
    uint8     fVolume;
    bool      fPowerState;
};

```

The code bears a striking resemblance to a `struct` definition. There are a lot more methods than properties and a couple of odd words – `public` and `private`. The bits closely related to the operation of the television, such as `ConnectToSignal()`, are in the `private` section of the class definition and can only be called by the `Television` class' other methods.

It might seem just a little strange that there are six methods devoted to settings and getting the value of the `Television`'s properties. Wouldn't it be easier – and less work – to just make `fChannel`, `fVolume`, and `fPowerState` public? Yes, but it would make the code that works with it messier. Our objects should expose their internal workings only when absolutely necessary. This makes it possible to change how it operates internally without disturbing its interaction with the outside world. For example, if we decided to rename `fPowerState` to `fIsTurnedOn`, the name of the variable would need to be changed in a handful of places in the code. All of those places would be in the file which contains the code for the class' methods. However, if we had made `fPowerState` a public property, we would have to change every reference to it in the entire project. If your project is large, this is a *lot* more work. Hiding properties using methods is called **data abstraction**, a term commonly heard in C++.

Calling an object's methods requires you to have access to the object. The dot and arrow operators that we used to get to a structure's variables are used with classes to access its methods and properties.

```

int
main(void)
{
    Television tv;

    // Watching a TV that is turned off isn't much fun
    if (!tv.IsTurnedOn())
        tv.TogglePower();

    // Watch the Channel 8 news :)
    tv.SetChannel(8);

    return 0;
}

```

The other oddballs in the class definition are the functions `Television()` and `~Television()`. They don't have a return type – not even `void`! This is because they are special methods. `Television` is called the class' **constructor**, which is a method called whenever we create a `Television` object. `~Television()` is the class' **destructor**, which is called whenever we destroy a `Television` object. They are optional, but almost every class has a constructor and many have a destructor. They also play a key role when allocating and freeing objects. Speaking of which, let's look at the C++ way of using heap memory. We're going to need it very soon.

C++ Memory Allocation: new and delete

To create a new Television object, we don't call `malloc()`. In fact, you will hardly ever use it unless you are writing a program in C. Instead, we will be using `new` and `delete`, the C++ counterparts to `malloc()` and `free()`. It's best we look at how they are used in this code example.

```
int
main(void)
{
    // Create a pointer to hold our TV and an object to go with it. Using new
    // allocates enough memory to hold one Television object and then calls
    // its constructor.
    Television *tv = new Television();

    // Like before, it works better when it is turned on
    if (!tv->IsTurnedOn())
        tv->TogglePower();

    // The news on 8 is all bad. Maybe Phineas and Ferb is on?
    tv->SetChannel(172);

    // Free our TV. Right before its memory is freed, the destructor method for
    // the Television class is called.
    delete tv;

    return 0;
}
```

It's much simpler to use `new` and `delete` for allocating memory on the heap. In fact, for objects, they are required. Other methods will cause an object's constructor and/or destructor to never be executed and lead to all kinds of ugly messes. `new` and `delete` can also be used for arrays of objects, as you can see in this code example.

```
int
main(void)
{
    // Create an array of TVs. I guess we're going to open an appliance store,
    // starting with 100 sets. We're going to need a little retail space. ;-)
    Television *tvArray = new Television[100];

    // Like before, it works better when it is turned on, but we'll turn just
    // one on, seeing how they're all the same. ;-)
    if (!tvArray[0].IsTurnedOn())
        tvArray[0].TogglePower();

    // We don't care what channel is on so long as we can't hear it.
    tvArray[0].SetVolume(0);

    // Free our TV. The brackets are needed when freeing an array we received
    // from new. Leaving out the brackets will cause a memory leak because
    // then only one of the objects will be freed instead of all of them.
    delete [] tvArray;

    return 0;
}
```

Construction and Destruction

The two special functions that we have seen in the definition for our `Television` class are part of the C++ language itself. The main job of a constructor function is to do whatever is necessary to initialize the object. When we allocate a `struct`, its variables contain random data. It's the same way with objects, but the constructor does the initialization for us.

A class has a default constructor if one is not declared in the class definition. The default constructor for an object takes no parameters and does nothing. Our `Television` class defines the default constructor, but this is not a requirement for a class. Doing so, however, replaces the one provided and does something. The constructor for a class can also take parameters, such as either of these possibilities or something else:

```
Television(const char *name);  
Television(bool isHD);
```

The destructor for a class always looks the same: a tilde (~), the name of the class, and taking no parameters. It is provided if a destructor is not declared in a class' definition. Like the default constructor, it doesn't do anything. The main job for a destructor is to clean up before the object is actually freed. Most of the time, this means freeing heap memory that was allocated somewhere in the object's methods.

Assignment

Practice thinking about object-oriented programming by writing down what a sample class definition for the following objects might look like, including both methods and properties:

- Alarm clock
- Car
- Stove
- Washing machine

Learning to Program with Haiku

Lesson 13

Written by DarkWyrn



In the last lesson, we learned about a programming paradigm that focuses on the design, construction, and interaction of objects to perform one or more tasks. We also examined how the objects in C++ are called classes, what their parts are, and some other fundamentals, but to be a proficient Haiku developer, we must learn more.

Inheritance

Much of the code a Haiku developer writes relates to **inheritance**, which is creating a class that uses another as its base. For example, let's say we have a Rectangle class that has height, width, and provides a couple of functions for area and perimeter. It would be easy to create a RectangularSolid class which just added a third dimension and a function for volume.

If the RectangularSolid class' third dimension is its only difference from a regular Rectangle, it seems like a lot of needless work to write code to calculate area and perimeter for both classes.. It would be great if we could just write code for only the parts that are different. Inheritance makes this possible: the RectangularSolid class is called a child class or a subclass of the Rectangle class. Just as a child inherits genes from his parents, a subclass inherits methods and properties from its parent class, and our RectangularSolid class inherits all of the Rectangle's properties and methods.

| | Rectangle | RectangularSolid |
|------------|-------------------|---|
| Properties | Height Width | Height (inherited) Width (inherited) Depth |
| Methods | Area Perimeter | Area (inherited) Perimeter (inherited) Volume |

Inheritance allows us to reuse code that we've already written. Code reuse is foundational to C++ programming. Once again, work smarter, not harder.

To turn this example into code, we will write two class definitions, one for each class.

```
class Rectangle
{
public:
    Rectangle(int width, int height);

    void SetWidth(int width);
    int Width(void);

    void SetHeight(int height);
    int Height(void);

    int Area(void);
    int Perimeter(void);

private:
    int fHeight;
    int fWidth;
};
```

```

// This line is what makes RectangularSolid a subclass of Rectangle
class RectangularSolid : public Rectangle
{
public:
    RectangularSolid(int width, int height, int depth);

    // We don't list the methods inherited from Rectangle – only the ones
    // which are new to the child class
    void SetDepth(int depth);
    int Depth(void);

    int Volume(void);

private:
    int fDepth;
};

```

The RectangularSolid class only declares methods and properties that are new. Writing the code that goes with these classes requires more than what the class definitions would make you think. This is our first code working with classes, so study this code and the comments carefully.

```

// Rectangle's constructor. This will set the object's properties to what were
// passed to it. When writing the code for a class' methods, the class name plus
// two colons must precede the method's name.
Rectangle::Rectangle(int width, int height)
{
    fWidth = width;
    fHeight = height;
}

void Rectangle::SetWidth(int width)
{
    fWidth = width;
}

int Rectangle::Width(void)
{
    return fWidth;
}

void Rectangle::SetHeight(int height)
{
    fHeight = height;
}

```

```
int
Rectangle::Height(void)
{
    return fHeight;
}
```

```
int
Rectangle::Area(void)
{
    return fWidth * fHeight;
}
```

```
int
Rectangle::Perimeter(void)
{
    return (2 * fWidth) + (2 * fHeight);
}
```

// This is the RectangularSolid constructor. In creating the RectangularSolid, it first creates a Rectangle object using the Rectangle's constructor function. We pass width and height to it and then use depth to initialize fDepth

```
RectangularSolid::RectangularSolid(int width, int height, int depth)
    : Rectangle(width, height)
{
    fDepth = depth;
}
```

```
void
RectangularSolid::SetDepth(int depth)
{
    fDepth = depth;
}
```

```
int
RectangularSolid::Depth(void)
{
    return fDepth;
}
```

```
int
RectangularSolid::Volume(void)
{
    // We have to call Width() instead of using fWidth and Height() instead of
    // fHeight because even child classes can't access an object's private
    // methods and properties.
    return Width() * Height() * fDepth;
}
```

The real difference between this code and all of the code we've written up to this point is that thinking in terms of objects forces us to organize our code. Writing neat and organized code is imperative to making maintaining it easier.

The only part of this code that might seem strange is the need to call `Width()` and `Height()` in `Volume()`. It seems like it would make more sense to allow child classes to be able to access `fWidth` and `fHeight`. That is what `protected` access is for. It's not needed or used nearly as much as you might think, though.

One other point to note is the line which handles the inheritance – it reads `public Rectangle`. This is the type of inheritance. Almost all of the time you will use `public` inheritance. This means that the access classes of the parent's methods and properties do not change. Choosing one of the other two types limits the access to the base class. Choosing `protected` inheritance will make all `public` methods and properties of the base class `protected` instead of `public`, making them available to all child classes, but closed to the outside. `Private` inheritance makes all methods and properties of the base class `private`, making the parent class totally inaccessible to both the outside world and to any "grandchild" classes.

Virtual Functions

Not only is it possible for a child class to add new methods and properties, but it can also change the behavior of existing methods. This is possible only when the base class says that doing so is allowed. Adding the `virtual` keyword before the return type in a method declaration grants this permission.

```
// A child class can redefine the behavior of this method
virtual void MyChangeableMethod(int someInt);

// A child class is required to define this method
virtual void ThisMethodMustBeDefined(float someFloat) = 0;
```

Being able to change how a method works does **not** enable us to change the number or types of a method's parameters or its return type, however. The original version found in the parent class doesn't disappear completely either. It can be specified using the scope operator, as exemplified below:

```
void
ChildClass::DoSomething(void)
{
    printf("Child class did something\n");
    ParentClass::DoSomething();
}
```

Static Functions

You usually have to have an instance of an object to call one of its methods. This can be a hassle sometimes, but declaring a method as `static` binds the method to the class, removing this requirement. `Static` methods are called using the scope operator the same way that we did for calling the parent implementation of a virtual function above.

```
class MyClass
{
public:
    MyClass(void);
    int DoSomething(void);
    static int DoSomethingStatic(void);
};
```

```

int
main(void)
{
    MyClass myClassInstance;

    // The regular way of calling a method
    myClassInstance.DoSomething();

    // An instance not required for this one – just a slightly different way to
    // call the method
    MyClass::DoSomethingStatic();
    return 0;
}

```

Overloading: Functions with the Same Name

One limitation of programming in C that can be a real pain in the neck is that no two functions can have the same name even if they have different parameters. C++ removes this limitation so long as the compiler is able to figure out which version is being called based on the number and types of arguments given to it. This will work:

```

int MyFunction(int oneWay);
int MyFunction(char *anotherWay);
int MyFunction(float aThirdWay);

```

This, on the other hand, won't work:

```

int SomeMethod(const char *oneConstString);
int SomeMethod(const char *anotherString, const char *optionalString = NULL);

```

Why doesn't this work? If you leave out the `optionalString` parameter, the compiler can't figure out which one you mean.

Assignment

Read through the sections of the BeBook on `BApplication`, `BWindow`, and `BView`. You don't have to understand everything, but try to get a feel for each class – in the next lesson we will be writing our first *real* program for Haiku.

Learning to Program with Haiku

Lesson 14

Written by DarkWyrn



If you've been waiting excitedly for the chance to learn how to write a program that shows something on the screen instead of just printing stuff on the Terminal, today's the day! We will spend most of our time understanding the code, but let's talk about tools and organization first.

Tools of the Trade: Integrated Development Environments

All of our programs so far have been simple. Console programs, as they are called, don't have to be, but to learn the language, simple is good. A text editor is all we've needed to do these simple programs which can all fit into one file. Regular applications can have hundreds of files. Many experienced C++ programmers are content with a text editor and Terminal, but having more isn't a bad thing, especially for novice coders.

For the rest of these lessons, you will be encouraged – but not required, however – to use an Integrated Development Environment (IDE). IDEs tie together the compiler, the code editor, the debugger, and other tools so that a developer can work more efficiently. For the purposes of these lessons, use of the Paladin IDE for Haiku will be assumed. If you prefer using another one, that's just fine. Let's get started.

Our First Project

Run Paladin and you will be greeted by the start window. Choose to create a new project. At the top of the New Project window is a menu where you can choose what kind of project to create. Choose Empty Application. While we could choose GUI with Main Window to automate what we are about to do, the act of typing the source code will help us learn the basics of Haiku GUI programming better. Set your project name to whatever you like. The Target Name is the name that the executable will have when your project is built and is often the same as the project name.

When you're ready to move on, click Create Project.

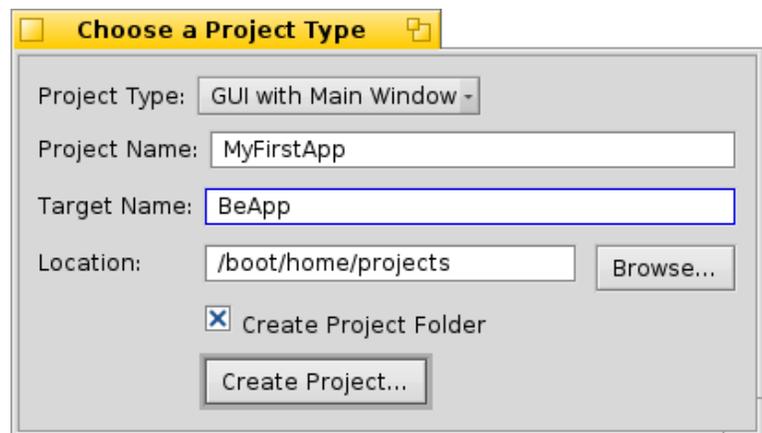


Illustration 1: The Paladin New Project window

As of this writing, if you are using a GCC4-based build of Haiku, you will need to add a library to your project to be able to build it properly. In your project's window, choose Change System Libraries from the Project menu. Scroll down and check the box beside libsupc++.so. Close the window and from the Project menu, choose Add New File. When asked the name of your file, enter App.cpp, check the box to create the corresponding header file, and click OK.

In general, there is one class per pair of files and the base of the files' names is the same as the class that goes in it. Our first file, App.cpp will have one class: App. The definition for the App class will go into the header file App.h. All of the code for the class will go into App.cpp. If we had a couple of really small related classes, it would be acceptable to put them both into the same file, but the general rule of thumb is one class per file pair for organization's sake.


```

// We'll reuse our BRect variable to set the location of our label. The
// actual width and height don't actually matter, though.
frame.Set(10,10,11,11);

// Create a static text label which has the text "Haiku Rocks!"
BStringView *label = new BStringView(frame,"mylabel","Haiku Rocks!");

// Tell the label to resize itself to fit the text that we've given it,
// saving us a lot of work in having to figure it out ourselves.
label->ResizeToPreferred();

// Attach the label to our window
myWindow->AddChild(label);

// Show our window
myWindow->Show();
}

int
main(void)
{
    // Create the instance of our App class. Each Haiku program has just one.
    // Creating it sets up the connection to the app_server.
    App *app = new App();

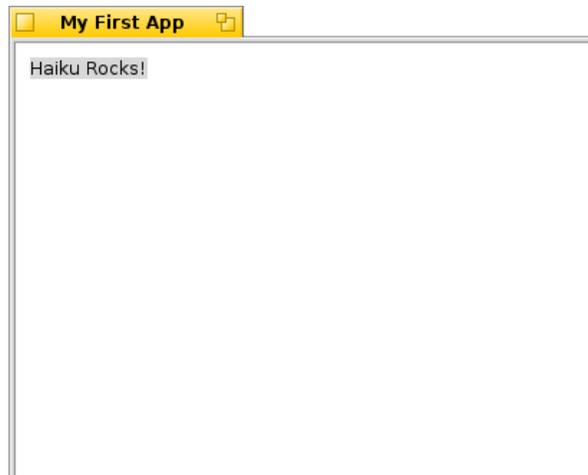
    // This actually puts our application into motion. We will not be exiting
    // this function until our program quits.
    app->Run();

    // Free the memory that we got off the heap. We could create it on the
    // stack, but BApplication objects can get kinda big, so it's better to use
    // the heap.
    delete app;

    // Token return call to quiet the compiler. ;-)
    return 0;
}

```

Save your work and either click on Run from the Build menu or press Alt + R to build and run your project. If you've typed everything correctly, you should see this on the screen:



If you run into errors, carefully check to make sure that you've typed everything just the same as the code above.

We've written our first working Haiku app! Not only does it show something on the screen, but we can resize the window, hide it, or close it and quit the program. It's not terribly useful, but that's OK. We had to learn a lot just to get this far and still understand what's going on.

One small bit of code was not explained in the comments, namely these two lines:

```
App::App(void)
    : BApplication("application/x-vnd.dw-MyFirstApp")
```

The part inside the quotation marks doesn't seem at all familiar. This is what's called a MIME type. MIME stands for **M**ultipurpose **I**nternet **M**ail **E**xtensions. Every file in Haiku has a type, including all Haiku programs. Graphical Haiku programs like ours have a unique type. Substitute your company's name, your online handle, or your initials for the dw and the name of your program for MyFirstApp – just make sure that the signature begins with "application/x-vnd." This will ensure no mixups with any programs you publish online.

This project was a barebones application to show the minimum code needed to create a simple window. It might seem like a lot, but in comparison to a Windows program written in C++, it's quite short. Next time, we'll make a window that does more, but for now, play around with the code and see what kinds of things you can do with it – tinkering leads to better understanding.

Learning to Program with Haiku

Lesson 15

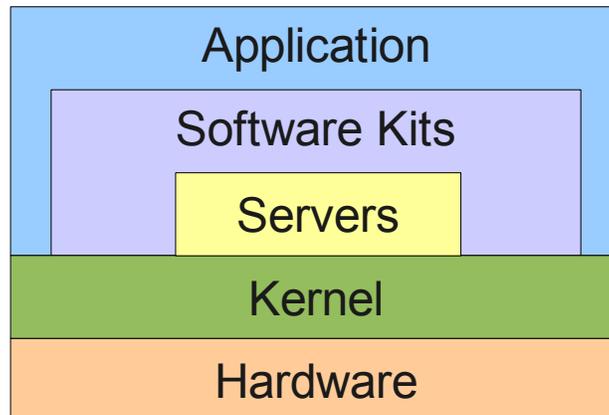
Written by DarkWyrn



Now that we've written our first working – albeit simple – GUI application for Haiku, we will start learning the fundamentals of what is considered the Haiku API.

Overview of the Haiku API

All of the operating system libraries that are available to us are organized into categorized groups called kits. Some of these kits have a server associated with them. Some do not. The majority of Haiku programs work with the kits and little else, although device drivers work directly with the hardware and mostly call kernel functions. A layout of Haiku as an operating system and its "layers" looks a little like this:



As of this writing the official kits are the following:

- Application
- Device
- Game
- Interface
- Kernel
- Mail
- Media
- MIDI
- Network
- OpenGL
- Storage
- Support
- Translation

In addition to these official kits, there are also two other which are under development and considered experimental: the Layout Kit and the Locale Kit. If this seems like a lot, it's because it is, but depending on the kinds of programs that you write you may never have to deal with certain kits. We'll look at some of the kits in detail later on, but for now, we will focus primarily on the Application, Interface, and Support kits.

Application Kit

The Application Kit is small, but vital. The focus centers around `BMessage`, the means for communication within a program and between programs, and `BApplication`, which must be subclassed to write a program which uses `BMessages`¹.

¹ This is actually a lie, but let's pretend it's not for now. It's easier that way.

Device Kit

The Device Kit provides classes for certain hardware. Because there are only two classes in this kit, it is not generally used, but it may expand in future versions of Haiku.

Game Kit

The Game Kit goes with the assumption that game programmers pretty much want to be given direct access to the section of memory used to display the screen – the video buffer – and then left alone to work their dark arts. Also here are some classes to make playing game sounds simple for game writers.

Interface Kit

To an applications programmer, the Interface Kit is as important as it is big. Windows, buttons, checkboxes, and more appear under its umbrella. Printing is also handled by this kit.

Kernel Kit

The Kernel Kit is the only kit which is not a collection of C++ classes. Instead, it consists of low-level C function calls.

Mail Kit

The Mail Kit is for constructing and sending e-mails. There's not very much else to say about it.

Media Kit

The Media Kit is all about audio and video processing. Haiku is especially good at playing and recording audio and video with a minimum of **latency**, or lag.

MIDI Kit

MIDI stands for **M**usical **I**nstrument **D**igital **I**nterface, a standard established in the ages of yore which defines how to get musical instruments to communicate with computers. The MIDI kit handles MIDI data just as the Media Kit is all about video and audio processing.

Network Kit

If you come from a UNIX-based programming background, you are probably accustomed to using C function calls for networking. The Network Kit's classes approach communications coding from a slightly friendlier perspective.

OpenGL Kit

If you're into 3D graphics, this is the kit for you. There is only one class, BGLView, but it opens the door to incorporating OpenGL graphics into your applications.

Storage Kit

The Storage Kit provides friendly ways of working with the filesystem. In addition to reading and writing files, there are also classes for reading directories, running queries, and working with attributes.

Support Kit

This kit is designed to support other kits with helper classes. BString, BList, and BLocker are especially helpful and are very commonly used.

Translation Kit

The Translation Kit is one of Haiku's innovations. It provides a single interface for reading and writing pictures and text without having to understand the underlying file formats. For us programmers, it makes life much, much easier.

Event-Based Programming

Writing a program for the console is pretty simple in that you, the developer, control the flow of execution. The same cannot be said for the GUI, with the reason that such programs are an interaction between the user and your program. The user does something and your program responds. Something strange happens in the system and your program alerts the user. Your code becomes a set of responses to different events. Much of this amounts to sending messages and handling any that come in.

An example of event-based programming is responding to the mouse. If the user clicks on a window's close button, the system will notify your program that the user is requesting the window to close. It's your job to do something about the request. When the user clicks a button, it sends a message. Who gets the message and what is done in response is up to you.

Haiku Messaging

A great deal of the communication that takes place within Haiku as an operating system centers around sending and handling messages. Most of the classes in the Application Kit center around messaging. Even though we won't use all of these right away, let's just take a quick peek at each of the classes in the kit:

- BApplication – The application class. It is also the main channel for communications between your program and the rest of the system.
- BClipboard – BClipboard handles storing information on a clipboard. The clipboard itself uses the BMessage class for storing and exchanging data with programs.
- BCursor – Not related to messaging, but BCursor takes care of changing what the mouse pointer looks like.
- BHandler – A class which is used to take appropriate actions for messages.
- BInvoker – A message-sending class used for controls such as buttons and checkboxes. Give it a message to send and a target to send messages to and it will send a copy of the message given to it each time its Invoke () method is called.

- BLooper – BLooper receives messages and passes them through a series of BHandlers before handling a message. It might sound confusing now, but it won't later on.
- BMessage – The type of object sent around the system for communications. It has an identifier property, what, and methods for attaching and retrieving data and for sending replies.
- BMessageFilter – A class used for filtering out desired – or undesired – messages.
- BMessageQueue – BMessageQueue stores messages in a first-in, first-out fashion. It is primarily used by BLooper instances to temporarily hold messages while it is handling others.
- BMessageRunner – This class sends messages at a specified interval.
- BMessenger – BMessenger is a message-sending class. It can send messages to BHandlers and BLoopers regardless of whether they are in your program or in another one.
- BPropertyInfo – Scripting is the purpose behind BPropertyInfo. If you're not enabling scripting your program from outside, you won't need this one often, if ever.
- BRoster – The BRoster class communicates with the system's application roster daemon. It is used for sending messages to all programs running in the system, launching programs, or for checking if a particular program is running.

Of all of these classes, the ones that are used in the course of regular applications programming are BLooper, BInvoker, BMessage, BHandler, and BApplication, so what seems like a lot to remember isn't really very much, especially when you consider that only a few methods of each of these classes are used frequently.

Reacting to most events in Haiku programming boils down to one function: `MessageReceived()`. It is a **hook function**, that is, a virtual function intended to be implemented by child classes to react to an event. In this case, `MessageReceived()` is implemented by child classes to handle messages that are not already handled by the parent class. Any child class of BHandler, including BLooper, BApplication, BWindow, and BView, have this hook function. Most of the time, it will look like this:

```
void
MyWindow::MessageReceived(BMessage *msg)
{
    switch (msg->what)
    {
        case M_SOME_MESSAGE:
        {
            DoSomething();
            break;
        }
        default:
        {
            // This calls the version of MessageReceived implemented by
            // MyWindow's parent class, BWindow.
            BWindow::MessageReceived(msg);
            break;
        }
    }
}
```

`MessageReceived()` can end up handling many different message codes, so a `switch` statement is called for here, and the `switch` differentiates between messages using the `what` identifier. Calling the

BWindow version of `MessageReceived()` is important because it handles all the messages that are ignored by the version that we have written.

Understanding how messaging works in Haiku is best learned in code, so we'll look at a second example, very much similar to the one from the last lesson, but which expands on what we know. We will create a window with a button. Clicking the button will change the title of the window to show the number of times the button has been clicked since the program was started. First, let's look at our window class. All of the code here can be found in the file `15ClickMe.zip`, but it would still be best to manually type out all of this code to increase your familiarity with it.

MainWindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>

class MainWindow : public BWindow
{
public:
    MainWindow(void);

    // We are implementing the virtual BWindow method MessageReceived so that we
    // can handle the message that the button will send to the window
    void MessageReceived(BMessage *msg);

private:
    // This property will hold the number of times the button has been clicked.
    int32 fCount;
};

#endif
```

MainWindow.cpp

```
#include "MainWindow.h"

// Button.h adds the class definition for the BButton control
#include <Button.h>

// The BView class is the generic class for creating controls and drawing things
// inside a window
#include <View.h>

// The BString class is a phenomenally useful class which eliminates almost all
// hassle associated with manipulating strings.
#include <String.h>

// This defines the identifier for the message that our button will send. The
// letters inside the single quotes are translated into an integer. The value for
// M_BUTTON_CLICKED is arbitrary, so as long as it's unique, it's not too
// important what it is. Note that we could use a #define for the message
// constant, but using an enum is the better way to go.
enum
{
    M_BUTTON_CLICKED = 'btcl'
```

```

};

MainWindow::MainWindow(void)
    : BWindow(BRect(100,100,300,200), "ClickMe", B_TITLED_WINDOW,
              B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE),
      fCount(0)
{
    // Create a button in pretty much the same way that we did the label in
    // the last lesson. The BRect() call inside the BButton constructor is a
    // quick shortcut that eliminates having to create a variable.
    BButton *button = new BButton(BRect(10,10,11,11), "button", "Click Me!",
                                  new BMessage(M_BUTTON_CLICKED));

    // Like with last lesson's label, make the button choose how big it should
    // be.
    button->ResizeToPreferred();

    // Add our button to the window
    AddChild(button);
}

void
MainWindow::MessageReceived(BMessage *msg)
{
    // The way that BMessages are identified is by the public property 'what'.
    switch (msg->what)
    {
        // If the message was the one sent to the window by the button
        case M_BUTTON_CLICKED:
        {
            fCount++;

            BString labelString("Clicks: ");

            // This converts fCount to a string and appends it to the end of
            // labelString. More on this next lesson.
            labelString << fCount;

            // Set the window's title to the new string we've made
            SetTitle(labelString.String());
            break;
        }
        default:
        {
            // If the message doesn't match one of the ones we explicitly
            // define, it must be some sort of system message, so we will
            // call the BWindow version of MessageReceived() so that it can
            // handle them. THIS IS REQUIRED if you want your window to act
            // the way that you expect it to.
            BWindow::MessageReceived(msg);
            break;
        }
    }
}

```

The main part of this program centers around the `M_BUTTON_CLICKED` case. When our button is added to the window, it sets the window as the target for its messages so that every time the button is clicked the window will receive a `M_BUTTON_CLICKED` message. When the window receives the button's message, it increments the member variable `fCount` and uses it to generate a new title.

Creating the title isn't difficult, especially if we use the `BString` class. The C way of doing it would be by allocating a string big enough to hold the title and then using `sprintf()`, but `BString` was designed to make working with strings in C++ a lot easier. Memory allocation is handled for us, and there are methods which combine strings, return the length, and much more. The `labelString << fCount` converts `fCount` into a string and tacks it onto the end of the string held by `labelString`.

The rest of the code kept in `App.h` and `App.cpp` is almost exactly the same as it was in the last lesson. The main difference is that `App.cpp` includes `MainWindow.h`. By including it, we have the definition for the `MainWindow` class and we can allocate and show one.

Going Further

Here are some possible changes you might like to explore to make this program do more. I would encourage you to try some or all of these changes. Experimentation leads to many "Aha!" moments and better programming.

- Change the numbers in the `BRect()` used to create the button and disable the `ResizeToPreferred()` call to make the button really, really big – almost as big as the window itself.
- Move the button to one of the window's corners
- Add a second button which sends a `B_QUIT_REQUESTED` message to the window to make it close.
- Create several buttons which move the window. (Hint: use a different message ID for each, and call `BWindow's MoveBy()` method in `MessageReceived()`)

Learning to Program with Haiku

Lesson 16

Written by DarkWyrn



What with all of the emphasis on getting the hang of making Haiku applications the last couple of lessons, you might be getting the impression that we're done learning C++, but there are some more aspects of the language still waiting to be learned. We will address them as they are needed while learning to write Haiku programs.

Function and Operator Overloading

Normally you think of overloading as being something you typically don't want to do, like overloading an electrical circuit. In C++, it's something that we do all the time to extend the usefulness of an existing function or language operator. **Function overloading** defines several versions of a function which take different sets of parameters. Operators are just functions, so things like plus and minus signs can be extended to better integrate the classes we define with the rest of the API. This gives us the ability to make our objects incredibly flexible and convenient.

In last lesson's project we saw our first example of operator overloading with the BString class when we generated the title for the window. It looked like this:

```
BString labelString("Clicks: ");  
  
// This adds the value of fCount to the end of labelString. More  
// on this next lesson.  
labelString << fCount;
```

Normally the << operator does a binary shift left, but it works differently on a BString. The << operator has been overloaded to work as a more flexible shortcut for its Append() method, converting fCount to a string and then tacking it on to the end of labelString. Any operator in C++ can be overloaded, including array brackets, parentheses used for function calls, and the arrow operator.

Like many other aspects of C++, having the ability to do something does not mean that you should do it, however. Making the + operator perform subtraction can only lead to weeping and gnashing of teeth, for example. Except in rare cases, overloaded operators should perform pretty much the same operation as they do on other objects. In doing so, you will prevent confusion and headaches.

Except for a select few, operator functions may be implemented either as methods, i.e. part of a class, or regular functions. There are benefits and drawbacks to each. For example, here are the two ways that a + operator may be overloaded:

Method:

```
MyClass operator+(const MyClass &from);
```

Regular function:

```
MyClass operator+(const MyClass &first, const MyClass &second);
```

All other binary operators follow this format. As a general rule, make binary operators regular functions so that you can use the associated objects without having to explicitly typecast them. The only operators that must be methods are assignment ('='), function call ('()'), subscript ('[]'), and member selection ('->'). Other assignment operators, such as += and /=, should be methods, but are not required to be.

Unary operators follow this format for implementation:

Method:

```
bool operator!(void) const;
```

Regular function:

```
bool operator!(const MyClass &target);
```

There are also a couple of special case operators which deserve some attention. First of all, what about the ++ and -- operators? Each of them can be used two different ways. This means that there are two different ways to overload them. When overloaded, both the prefix (++i) and postfix (i++) versions must be implemented.

```
class MyClass:
{
    // Preincrement operator
    MyClass operator++(void);

    // Postincrement operator. The integer is just a dummy argument to
    // differentiate the two
    MyClass operator++(int dummy);
};
```

The subscript operator ('[]') also is a special case because it can be on either side of an assignment. It is required to be a method, so there is only one way to implement it:

```
MyClass & operator[] (const int index);
```

As a convenience to keep all of this hard-to-remember information, here is an almost-exhaustive table which organizes it into one place.

| Operator | Method | Regular | Operator | Method | Regular |
|----------|-------------|-------------|----------|-------------|-------------|
| + | | Recommended | = | Required | |
| - | | Recommended | += | Recommended | |
| * | | Recommended | -= | Recommended | |
| / | | Recommended | *= | Recommended | |
| % | | Recommended | /= | Recommended | |
| ++ | Recommended | | %= | Recommended | |
| -- | Recommended | | < | | Recommended |
| [] | Required | | <= | | Recommended |
| ^ | Recommended | | > | | Recommended |
| ~ | Recommended | | >= | | Recommended |
| ! | Recommended | | == | | Recommended |
| && | | Recommended | != | | Recommended |

| Operator | Method | Regular | Operator | Method | Regular |
|---------------|-------------|-------------|-----------|-------------|-------------|
| | | Recommended | Bitwise & | | Recommended |
| << | | Recommended | | | Recommended |
| >> | | Recommended | <<= | Recommended | |
| () | Required | | >>= | Recommended | |
| [] | Required | | &= | Recommended | |
| Reference & | Recommended | | = | Recommended | |
| Dereference * | Recommended | | ^= | Recommended | |
| -> | Required | | | | |

Copy Constructors

Having learned about constructors and destructors in the last lesson, we've learned about most of the basics, leaving out one related item: the copy constructor, but we're going to address it along the way as we deal with a problem with floating point numbers.

The `float` type isn't terribly accurate. Add together 50 or 100 floating point numbers and you won't necessarily have exactly the result you would have if you were to add them up with a calculator. This stems from how they are stored in memory. Normally, this isn't a problem, but if we were going to write a personal finance program, any rounding error would be a major issue. We're going to create a type which is accurate to merely two decimal places and no errors. First, let's start with a basic class definition and a quick `main()` function to test it out.

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

class Fixed
{
public:
    Fixed(void);
    ~Fixed(void);
    float GetValue(void);
    void SetValue(const int64 &value);

private:
    int64 *fValue;
};

Fixed::Fixed(void)
{
    fValue = new int64();
    *fValue = 0;
}

Fixed::~~Fixed(void)
{
    delete fValue;
}
```

```

}

float
Fixed::GetValue(void)
{
    return float(*fValue) / 100.0;
}
void
Fixed::SetValue(const int64 &value)
{
    *fValue = value * 100;
}

int
main(void)
{
    Fixed f;
    f.SetValue(1234);

    printf("Value: %f\n", f.GetValue());
    return 0;
}

```

The code itself isn't terribly complicated. We have four public methods: the constructor, which allocates heap memory for `fValue` and initializes the value to zero, the destructor, which frees the heap memory for `fValue`, and methods to get and set the object's value.

The idea behind our fixed class is that we're going to use a regular integer to hold a floating point value to avoid any rounding errors. The lowest two digits are reserved for the fractional part, so we will have to multiply any outside numbers by 100 to be able to add them to our Fixed class and divide the value of our Fixed class by 100 to translate to proper values for the outside world. So far everything seems to work properly. Let's tweak our `main()` function a bit:

```

int
main(void)
{
    Fixed f1;

    if (f1.GetValue() == 0)
    {
        Fixed f2;
        f2.SetValue(1234);
        f1 = f2;
    }
    printf("Value: %f\n", f1.GetValue());
    return 0;
}

```

If compiled and run under Haiku, once again everything seems to be OK, but there are two devilishly hard-to-find little bugs in the works that will pop up only if our program gets more complicated: the address held by `fValue` in our variable `f1` is deleted twice and there is memory leaked. Double deletes are in some ways worse than memory leaks because they can potentially crash your program and the location of the crash rarely has much to do with the location of the problem. Worse yet, changing the

code can cause it to crash in a different place without any apparent reason.

The problem with this code is that the last line of the `if` block doesn't do exactly what we want it to do. What we want is to copy the *value*, but what is copied is the *pointer*. When `f2` goes out of scope at the end of the `if` block, its `fValue` is deleted. The problem is that `f1`'s `fValue` is pointing to the same address. Uh-oh. `f1`'s `fValue` is pointing to a memory location that is now invalid. Anything can happen when `f1` is deleted when our program exits. The memory allocated is also floating out in the ether, never to be deleted until the operating system cleans up our mess for us. If a problem like this should appear in a program under Ubuntu Linux, error information is printed out, but for Haiku, nothing happens. This is actually worse because we're not told about a problem which exists.

Fixing the bug, fortunately, is pretty easy once we understand it. The problem is that a shallow copy was performed. Shallow object copies dump the exact values of the properties of one object into another. This is fine so long as the objects' properties are allocated on the stack. Heap-allocated properties require a deep copy. This can only be done by implementing our own copy constructor and, while we're at it, overloading the assignment operator.

The copy constructor is a function called whenever an object needs to be duplicated. The default copy constructor performs a shallow copy, which is sufficient in many cases. This isn't one of them, however. We will add these two entries to the class definition:

```
Fixed(const Fixed &from);  
Fixed & operator=(const Fixed &from);
```

The implementation of these two functions will look like this:

```
Fixed::Fixed(const Fixed &from)  
{  
    fValue = new int64();  
    *fValue = *from.fValue;  
}  
  
Fixed &  
Fixed::operator=(const Fixed &from)  
{  
    *fValue = *from.fValue;  
}
```

This copies the values in the addresses kept in `fValue` instead of copying addresses themselves, thus performing a deep copy. All is right with the universe again for a little while longer. Whew!

Project

Let's go one step further and really flesh out this class. It may come in quite handy at some point in the future. Here are the declarations for some more operator overloading that would be useful to integrate our new class into the rest of the programming environment:

```
Fixed operator+(const Fixed &first, const Fixed &second);  
Fixed operator-(const Fixed &first, const Fixed &second);  
bool operator<(const Fixed &first, const Fixed &second);  
bool operator>(const Fixed &first, const Fixed &second);
```

```
bool operator<=(const Fixed &first, const Fixed &second);  
bool operator>=(const Fixed &first, const Fixed &second);  
bool operator!=(const Fixed &first, const Fixed &second);  
bool operator==(const Fixed &first, const Fixed &second);
```

Get some overloading practice by implementing and testing these regular functions. You might also want to see what other operators might come in handy.

Learning to Program with Haiku

Lesson 17

Written by DarkWyrn



With a minor diversion into operator overloading and copy constructors, we've been slowly learning how to put together a graphical application for Haiku. So far we have examined the boilerplate code used to start any Haiku program, event-based programming, and sending messages. Today we'll be putting what we know about messages to use to create a menu for a program.

Using Menus

We're going to learn about several classes this lesson: BMenu, BMenuBar, BMenuItem, and BView, but we will be learning about them as we put together another relatively simple application that we'll call MenuColors, which shows a window with a colored box that we can change by choosing a color from a menu. Let's get our project set up.

1. Create a new, empty project in Paladin. You can call the project and target MenuColors or any other name you like.
2. Under the Project menu, click Add New File.
3. Type in the file name App.cpp, check the 'Create a header and a source file' box, and click OK or press Enter. This will create App.cpp and App.h.
4. Do the same for MainWindow.cpp.

Now that we have our files set up, let's bang out some code. First, here is the code for App.h and App.cpp. It would be a good idea to type everything out instead of copying and pasting the text so that you get familiarized with the boilerplate code.

```
// App.h
#ifndef APP_H
#define APP_H

#include <Application.h>

class App : public BApplication
{
public:
    App(void);
};

#endif

// App.cpp
#include "App.h"
#include "MainWindow.h"

App::App(void)
    : BApplication("application/x-vnd.test-MenuColors")
{
    MainWindow *mainwin = new MainWindow();
    mainwin->Show();
}

int
main(void)
{
```

```

    App *app = new App();
    app->Run();
    delete app;
    return 0;
}

```

None of the code in these files is anything new. Note that because we haven't entered in the class definition for MainWindow in its header file, if you try to build your project right now, you get some errors. Don't worry – we'll have everything working in just a moment. Here is the code for MainWindow.h:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>

class MainWindow : public BWindow
{
public:
    MainWindow(void);
    void MessageReceived(BMessage *msg);
};

#endif

```

This, too, is familiar territory. All of the new code will be in MainWindow.cpp, but let's start with the boilerplate code for MainWindow and add to it bit by bit. The best way to write code is to add a little at a time, compile, and then test what you've written to help you find bugs.

```

#include "MainWindow.h"

#include <MenuBar.h>
#include <Menu.h>
#include <MenuItem.h>
#include <View.h>

MainWindow::MainWindow(void)
    : BWindow(BRect(100,100,500,400), "MenuColors", B_TITLED_WINDOW,
              B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE)
{
}

void
MainWindow::MessageReceived(BMessage *msg)
{
    switch (msg->what)
    {
        default:
        {
            BWindow::MessageReceived(msg);
            break;
        }
    }
}

```

Now that we have all of our baseline code in place, build it just to make sure that you haven't mistyped anything. If you get errors, double check with the sources above and fix the typos. Once everything builds properly, run it and assuming everything works right, close it and move on to the interesting stuff.

Adding Views

Window controls are all objects which are descended from the BView class. If you take a quick glance at /boot/develop/headers/os/interface/View.h, you'll see that it's a large, complicated class with a lot of methods. Luckily, we don't need to know very many at this point. The BView below will be our colored box. Add this code to the MainWindow constructor:

```
BView *view = new BView(BRect(100,100,300,200), "colorview", B_FOLLOW_ALL,
                        B_WILL_DRAW);
AddChild(view);
view->SetViewColor(0,0,160);
```

This is pretty simple code. The first line creates a new BView which has its top left corner at (100,100) and its bottom right corner at (300,200). It has the name "colorview" and resizes itself whenever the window is resized. The B_WILL_DRAW flag at the end tells the window that it does its own drawing. Without this flag, this BView will just be a blank white box. The second line attaches the view to the window. The last one sets the color of the BView to a dark blue color. Build your project and see how it all looks.

Adding controls to windows in Haiku isn't much different. Most of them have a constructor which requires the same kinds of information and possibly a message, but not much else. Other controls are added to a window just the same way. Pay close attention to the methods that we use and you will notice many similarities among the classes.

Adding a Menu

With the exception of pop-up menus, menus in Haiku are normally kept in a menu container of some sort. The Haiku API provides two menu containers: BMenuField and BMenuBar. We'll ignore BMenuField for now and focus on BMenuBar. Change the constructor code to the following:

```
// This will define the height of the menu bar. Bounds() returns the size of the
// window. In this case, the rectangle will be (0,0)-(200,100).
BRect r(Bounds());
r.bottom = 20;

// The only part of r that matters is the height. When we add items to the menu
// bar, it will expand to fill the width of the window at the height we specify.
BMenuBar *menuBar = new BMenuBar(r, "menubar");
AddChild(menuBar);

BView *view = new BView(BRect(100,100,300,200), "colorview", B_FOLLOW_ALL,
                        B_WILL_DRAW);
AddChild(view);
view->SetViewColor(0,0,160);
```

Now that we have a container for our color menu, we can create the menu itself. This will require three components: the menu, the items in the menu, and message identifiers for each menu item. First, let's handle the message identifiers. Add this code just after the `#include` statements at the top of the file.

```
enum
{
    M_SET_COLOR_RED = 'sred',
    M_SET_COLOR_GREEN = 'sgrn',
    M_SET_COLOR_BLUE = 'sblu',
    M_SET_COLOR_BLACK = 'sblk'
};
```

The only part of this code that might look strange are the single-quoted values. This is another one of those fancy-schmancy coder tricks. Message constants are 32-bit integers. Each one of those letters translates into an 8-bit value, so 'sred' actually translates into a 32-bit integer. Truth be told, the values themselves don't really matter much as long as they are unique, but by convention they are set to 4-letter constants like this.

The constants themselves can either be placed at the top of a class' source file or in its header. You will usually want to avoid putting the identifiers in the header so that adding a message identifier doesn't force a recompile of several files. The exception to this rule of thumb would be for messages that are used by more than one class.

Now that the message identifiers have been defined, let's go on to creating and populating the menu. The code looks like this:

```
MainWindow::MainWindow(void)
:    BWindow(BRect(100,100,500,400), "MenuColors", B_TITLED_WINDOW,
            B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE)
{
    BRect r(Bounds());
    r.bottom = 20;

    BMenuBar *menuBar = new BMenuBar(r, "menubar");
    AddChild(menuBar);

    // This is the code that creates and populates the menu.
    BMenu *menu = new BMenu("Colors");
    menu->AddItem(new BMenuItem("Red", new BMessage(M_SET_COLOR_RED), 'R'));
    menu->AddItem(new BMenuItem("Green", new BMessage(M_SET_COLOR_GREEN), 'G'));
    menu->AddItem(new BMenuItem("Blue", new BMessage(M_SET_COLOR_BLUE), 'B'));
    menu->AddItem(new BMenuItem("Black", new BMessage(M_SET_COLOR_BLACK), 'K'));

    // The menu bar adds menus the same way that a menu adds items. In fact, the
    // menu bar is more or less a menu whose items are arranged horizontally
    // instead of vertically.
    menuBar->AddItem(menu);

    BView *view = new BView(BRect(100,100,300,200), "colorview", B_FOLLOW_ALL,
                           B_WILL_DRAW);
    AddChild(view);
    view->SetViewColor(0,0,160);
}
```

We're almost done! If you try running your project and clicking in the menu, you'll find that nothing happens. Everything works the way it should. When a menu item is clicked, it sends a message to the window, like clicking on the 'Red' menu item sends a `M_SET_COLOR_RED` message to the window. The window receives it, but doesn't do anything with it, so all that is needed now is to write the code for when the window receives each of the menu items' messages.

```
void
MainWindow::MessageReceived(BMessage *msg)
{
    // FindView() is a BWindow method which searches for a BView by name and
    // returns a pointer to it.
    BView *view = FindView("colorview");

    switch (msg->what)
    {
        case M_SET_COLOR_RED:
        {
            // When the window receives this message, we'll set the
            // background color to dark red
            view->SetViewColor(160,0,0);

            // Calling Invalidate() forces the view to redraw itself.
            view->Invalidate();
            break;
        }
        case M_SET_COLOR_GREEN:
        {
            view->SetViewColor(0,160,0);
            view->Invalidate();
            break;
        }
        case M_SET_COLOR_BLUE:
        {
            view->SetViewColor(0,0,160);
            view->Invalidate();
            break;
        }
        case M_SET_COLOR_BLACK:
        {
            view->SetViewColor(0,0,0);
            view->Invalidate();
            break;
        }
        default:
        {
            // As always, the default is to make the BWindow version of
            // this method handle messages we don't care about
            BWindow::MessageReceived(msg);
            break;
        }
    }
}
```

We're done now! Run your project and you will see that our box changes color when you click on one of the menu's items. It really doesn't take much effort to put together a menu.

Going Further

Here are some ideas you can try if you'd like to tinker with it some more.

1) Try changing the `B_FOLLOW_ALL` resizing mode flag in the `BView` constructor to something else. Change it to one of these and see what it does:

- `B_FOLLOW_LEFT | B_FOLLOW_TOP`
- `B_FOLLOW_LEFT_RIGHT | B_FOLLOW_TOP`
- `B_FOLLOW_RIGHT | B_FOLLOW_TOP`
- `B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM`
- `B_FOLLOW_LEFT_RIGHT | B_FOLLOW_BOTTOM`
- `B_FOLLOW_RIGHT | B_FOLLOW_TOP_BOTTOM`

2) Try adding more colors to your menu

3) Add a Quit item to the menu which sends the window a `B_QUIT_REQUESTED` message.

Classes and Methods to Remember

BWindow

- `BWindow(BRect frame, const char *title, window_type type, uint32 flags, uint32 workspace = B_CURRENT_WORKSPACE)` – Create a new window. While there are other types, right now the window types to remember are `B_TITLED_WINDOW` and `B_DOCUMENT_WINDOW`.
- `void AddChild(BView *child)` – attaches a `BView` (or `BView` subclass) to the window.
- `BView * FindView(const char *name)` – Returns a pointer to a `BView` named `name` or `NULL` if not found.
- `BRect Bounds(void)` – Returns the size of the window's client area, i.e. the white area inside the window's frame.
- `void Show(void)` – Shows the window.

BView

- `BView(BRect frame, const char *name, int32 resizeMode, int32 flags)` – Create a new view. Check the `BView` section of the BeBook for all of the available resizing modes. Don't worry about the `flags` parameter just yet except to remember `B_WILL_DRAW`.
- `void SetViewColor(uint8 red, uint8 green, uint8 blue)` – Sets the background color of the view.
- `void Invalidate(void)` – Forces the `BView` to redraw itself.
- `void AddChild(BView *child)` – attaches a `BView` (or `BView` child class) to the view.

Learning to Program with Haiku

Lesson 18

Written by DarkWyrn



With buttons and menus out of the way, we will start learning more about some of the other kinds of window controls available to us, such as the following controls:

| Control | Description |
|------------------|--|
| BAlert | A pop-up message. They're great for the occasional error message, but they are also very easy to overuse and abuse. Use them sparingly. |
| BBox | A BView which draws a box around other child controls to visually group related controls together. It also can optionally have a text label in the top left corner. |
| BButton | A button with a text label which sends a message when clicked. |
| BChannelSlider | A slider class which supports multiple channels. It's mostly used with the Media Kit. |
| BCheckBox | A standard-issue checkbox which can send a message when clicked. |
| BColorControl | A customizable color picker. The main drawback is that it takes up quite a lot of space. |
| BListView | A list of items. The items themselves are sufficiently customizable to show just about anything but are often BStringItems to display a list of names. |
| BMenu | A container for menu items. |
| BMenuBar | A container for menus intended to be placed at the top of a window. |
| BMenuField | A button-like menu container which shows a pop-up menu. Getting them to resize properly can be problematic. |
| BMenuItem | A clickable item in a menu. |
| BOptionPopUp | A convenience class which quickly sets up a BMenuField. |
| BOutlineListView | A hierarchical list, but otherwise no different from BListView. |
| BPictureButton | A button which uses a BPicture vector graphic to draw a picture for a label. They can be one-state buttons like BButton or two-state buttons that stay down when clicked and pop back up when clicked again. |
| BPopupMenu | A special type of menu that can be shown anywhere on the screen. |
| BRadioButton | The circular "checkbox" that enables the user to choose one from several options. |
| BScrollBar | A single scrollbar used to scroll BViews. |
| BScrollView | A convenience class for attaching scrollbars to BViews and associated controls. |
| BSeparatorItem | A menu item which displays a horizontal line. They are used to separate groups of menu items. |
| BSlider | A highly-customizable horizontal slider. |
| BStatusBar | A progress bar class. |
| BStringView | A static text label. |

| Control | Description |
|--------------|--|
| BTabView | Group together sheets of controls on tabs. It requires a little more care to use them than what it might initially seem. |
| BTextControl | A single line text editing control. |
| BTextView | A multiline text editing control with limited word processing capabilities. |
| BView | The base class from which all controls are derived. It has a lot of methods and it can be a little overwhelming, but it's powerful and flexible. |

That's a lot of different classes! At first, it seems pretty overwhelming, but it's not too bad after a little tinkering, especially when you consider that there is quite a bit they have in common. You might want to keep this control table handy for your first projects. We're not going to look at every control in detail, but the ones that we skip should be easy enough to pick up on your own.

Typecasting Revisited

Way back in lesson 7 we learned about typecasting, i.e. reinterpreting the data in a variable as a different type. One of the problems with casting in C is that it isn't safe – whenever you cast a variable, it always succeeds. If you choose the wrong class to cast, it still succeeds, typically with unpredictable results. C++ offers different kinds of casting. The C++ way should be preferred unless you're writing code in C. Here they are:

```
static_cast<TypeToCastTo *>(pointerToCast);
```

Static casts are used to convert one type to another. It relies upon compile-time information and is often used for the purposes that the regular C casting method is used – changing pointer types and arithmetic conversions. It works so long as the language supports the conversion between the two types.

```
dynamic_cast<TypeToCastTo *>(pointerToCast);
```

Aside from static casts, dynamic casts will be one of your most commonly-used cast types when working with Haiku. They are used to safely navigate an inheritance hierarchy. If no inheritance links together the "from" type and the "to" type, NULL will be returned instead. Thus, you can cast to only those pointer types you're supposed to be able cast. We will use one of these in today's project.

```
const_cast<TypeToCastTo *>(pointerToCast);
```

Const casting adds or removes the changeability of a pointer. For example, there are instances where a function will take a non-const parameter that it doesn't change. Passing a constant pointer to one of these functions will require a const cast.

```
reinterpret_cast<TypeToCastTo *>(pointerToCast);
```

A reinterpret cast is used only rarely. It converts a pointer from one type to another, regardless of whether the two pointers are related or not. Almost all of the tasks which a reinterpret cast can do can be done with a static cast, and the remaining conversions are almost always not portable, so this kind of cast should be used only when absolutely necessary, such as converting between function pointer types.

Project: Using List Controls

This project, called ListTitle, will be about using list controls, namely the BListView class. One paradigm found in the Haiku API is that list-based controls use lightweight items. BMenu and BListView actually do most of the work and BMenuItem and BListItem are actually very simple classes. Designing these classes this way saves memory.

1. Create a new project in Paladin, but this time use the "GUI with Main Window" template to save some typing – this template creates the boilerplate code for the App and MainWindow classes.
2. Our project and target name will be ListTitle.
3. Once created, open App.cpp, change the MIME signature of the app to "application/x-vnd.test-ListTitle", and close it.
4. Open MainWindow.h, add the include ListView.h to the top of the header.
5. Add a private: access section keyword at the bottom of the MainWindow class definition.
6. In the private section of the MainWindow definition, declare the property BListView *fListView. We will be using fListView in more than just the window constructor, so it makes sense to stash away a pointer to it for later use.

Now let's get down to business: setting up the MainWindow's controls and making them do something. Open up MainWindow.cpp and change it to this code:

```
#include "MainWindow.h"

#include <Button.h>
#include <ListItem.h>
#include <ScrollView.h>

enum
{
    M_RESET_WINDOW = 'rsw'n',
    M_SET_TITLE = 'sttl'
};

MainWindow::MainWindow(void)
    : BWindow(BRect(100,100,500,400), "The Weird World of Sports",
            B_TITLED_WINDOW, B_ASYNCHRONOUS_CONTROLS |
            B_QUIT_ON_WINDOW_CLOSE)
{
    // Here we will make a BView that covers the white area inside the window so
    // we can choose a background color. You'll want to do this in the windows
    // of your projects -- your projects will look more professional
    BRect r(Bounds());
    BView *top = new BView(r, "topview", B_FOLLOW_ALL, B_WILL_DRAW);
    AddChild(top);

    // ui_color() returns a system color, such as the window tab color, menu
    // text color, and so forth. The Panel Background color is the one used
    // for background views like this one.
    top->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));

    // Create a button and place it at the bottom right corner of the window.
    // The BRect that we use for the BButton's frame is empty because we're
    // going to have it resize itself and then move it to the corner based on
```

```

// the actual size of the button, so it's pointless to specify a size
BButton *reset = new BButton(BRect(),"resetbutton","Reset",
                             new BMessage(M_RESET_WINDOW),
                             B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM);
top->AddChild(reset);
reset->ResizeToPreferred();

// Place the button in the bottom right corner of the window with 10 pixels
// of padding between the button and the window edge. 10 pixels is kind of a
// de-facto standard for control padding. It's enough that controls don't
// look crowded without taking up tons of space.
reset->MoveTo(Bounds().right - reset->Bounds().Width() - 10.0,
             Bounds().bottom - reset->Bounds().Height() - 10.0);

r = Bounds();
r.InsetBy(10.0,10.0);

// When working with BScrollViews, you must compensate for the width/height
// of the scrollbars when determining the size of the control that we will
// attach to the BScrollView. B_V_SCROLL_BAR_WIDTH is a defined constant for
// the width of the vertical scroll bar.
r.right -= B_V_SCROLL_BAR_WIDTH;

// Frame works like Bounds() except that it returns the size and location of
// the control in the coordinate space of the parent view. This will make
// fListView's bottom stop 10 pixels above the button.
r.bottom = reset->Frame().top - 10.0 - B_H_SCROLL_BAR_HEIGHT;

// Most of these parameters are exactly the same as for BView except that we
// can also specify whether the user is able to select just 1 item in the
// list or multiple items by clicking on items while holding a modifier key
// on the keyboard.
fListView = new BListView(r,"colorlist",B_SINGLE_SELECTION_LIST,
                          B_FOLLOW_ALL);

// We didn't call AddChild on fListView because our BScrollView will do that
// for us. When created, it creates scrollbars and targets the specified
// view for any scrolling they do. When the BScrollView is attached to the
// window, it calls AddChild on fListView for us.

// If we call AddChild on fListView before we create this scrollview, our
// program will drop to the debugger when we call AddChild on the
// BScrollView -- a BView can have only one parent.
BScrollView *scrollView = new BScrollView("scrollview",fListView,
                                         B_FOLLOW_ALL,0,true,true);
top->AddChild(scrollView);

// A BListView's selection message is sent to the window any time that the
// list's selection changes.
fListView->SetSelectionMessage(new BMessage(M_SET_TITLE));

fListView->AddItem(new BStringItem("Toe Wrestling"));
fListView->AddItem(new BStringItem("Electric Toilet Racing"));
fListView->AddItem(new BStringItem("Bog Snorkeling"));
fListView->AddItem(new BStringItem("Chess Boxing"));
fListView->AddItem(new BStringItem("Cheese Rolling"));

fListView->AddItem(new BStringItem("Unicycle Polo"));

```

```

}

void
MainWindow::MessageReceived(BMessage *msg)
{
    switch (msg->what)
    {
        case M_RESET_WINDOW:
        {
            fListView->DeselectAll();
            break;
        }
        case M_SET_TITLE:
        {
            int32 selection = fListView->CurrentSelection();

            if (selection < 0)
            {
                // This code is here because when we press the Reset
                // button, the selection changes and an M_SET_TITLE
                // message is sent, but because nothing is selected,
                // CurrentSelection() returns -1.
                SetTitle("The Weird World of Sports");
                break;
            }

            BStringItem *item = dynamic_cast<BStringItem*>(
                fListView->ItemAt(selection));

            if (item)
                SetTitle(item->Text());
            break;
        }
        default:
        {
            BWindow::MessageReceived(msg);
            break;
        }
    }
}
}

```

There's not really that much to this project that's terribly different from the last one. By calling the BListView's SetSelectionMode() method, we cause the title to be updated any time the user clicks on an item in the list. Most of the time we use a BListView we won't use this method. It's much more common to call a related one: SetInvocationMessage(), which sends a message whenever the user double-clicks on an item. Note that DeselectAll() also causes a selection message to be sent even though there isn't a selection, so it is necessary to handle the case when CurrentSelection() returns a negative value, signifying no selection.

Hopefully you're getting a feel for how BViews and regular controls are put together in applications. Most of them require a BRect for the size and location, a const char * for the name of the control, and two integers for the resizing mode and some behavior flags. Many classes also have a label and a message that is sent when the control is changed or invoked, especially those derived from BControl. Once a control is created, it is attached to the BWindow or a BView via AddChild(). The message sent by the control is often sent to the window to which the control is attached, but it can be redirected to

another target, such as the parent BView or to the global BApplication.

Classes and Methods to Remember

BControl

- `ResizeToPreferred(void)` – A derived class will resize itself to a good size to display its label and content.
- `SetLabel(const char *label) / const char * Label(void)` – Methods to get and set the label for a child class.
- `SetTarget(BHandler *handler, BLooper *looper)` – Send the invocation message to a different target, such as a BView, BWindow, or BApplication.
- `void SetEnabled(bool enabled) / bool IsEnabled(void)` – Methods to get and set the enabled/disabled status of a control.

BListView

- `AddItem(BListItem *item)` – Add an item to the list.
- `int32 CountItems(void)` – Returns the number of items in the list.
- `BListItem * RemoveItem(int32 index)` – Removes and returns the item at the specified index or NULL if there isn't one.
- `void RemoveItem(BListItem *item)` – Removes the specified item from the list. If the list doesn't have the item, it doesn't do anything.
- `int32 CurrentSelection(int32 index = -1)` – Returns the index of the currently selected item or -1 if there is no selection. The index argument is used to get all the selected items in a list which supports multiple item selections. A `while()` loop is generally used to get all the item indices and it normally exits when -1 is returned.
- `void Select(int32 index, bool extend = false)` – Select the item at the specified index. If `extend` is false, all other selected items are deselected before the specified item is selected.
- `void Select(int32 start, int32 end, bool extend = false)` – Selects all items from `start` to `end`. If `extend` is false, all other selected items are deselected before the specified items are selected.
- `void DeselectAll(void)` – deselects all items in the list.

Learning to Program with Haiku

Lesson 19

Written by DarkWyrn



The Interface Kit has been our primary source of functionality in the last couple of lessons, but today we'll be expanding our reach. This lesson will deal with the Translation Kit and part of the Storage Kit.

The Translation Kit

Originally written by Jon Watte in ancient Be history, the Translation Kit has nothing to do with language translation. Instead, it is used for converting images and text from one format to another. This usually amounts to loading bitmap images from disk. A **bitmap** is a kind of image which stores its picture information as a collection of dots. **Vector images** store their information as a series of drawing instructions. This is the difference between a photo, which is a bitmap image, and a clipart picture.

The Translation Kit uses a unique approach to loading pictures: its capabilities are defined and expanded by add-ons. Add-ons are a powerful programming approach used in many places in Haiku. What makes them so powerful in this case is that installing a Translator for a particular image format enables every Haiku program which utilizes the Translation Kit to suddenly support that format. It's a much better approach than can be found on any other operating system available. It also makes for much simpler image handling code.

There are two different ways that the Translation Kit can be used in our programs. Which one we should use in a given situation depends on our needs. The easy way is just for loading bitmaps. The more involved way is needed for instances when we are both loading and saving or when we are using the Translation Kit for something other than bitmap images. We'll be using the easy way today, the details of which we'll be seeing shortly.

Resources

Application resources are a way to package data associated with your program inside the executable itself. It's a handy way to make sure that pictures used in your program are always available and haven't been overwritten by the user's recipe for quiche or something. Icons for your program are resources. Pictures used in the GUI should be stored this way, too. Certain other program information, such as the file types it handles and launch characteristics, are also stored in a program's resources.

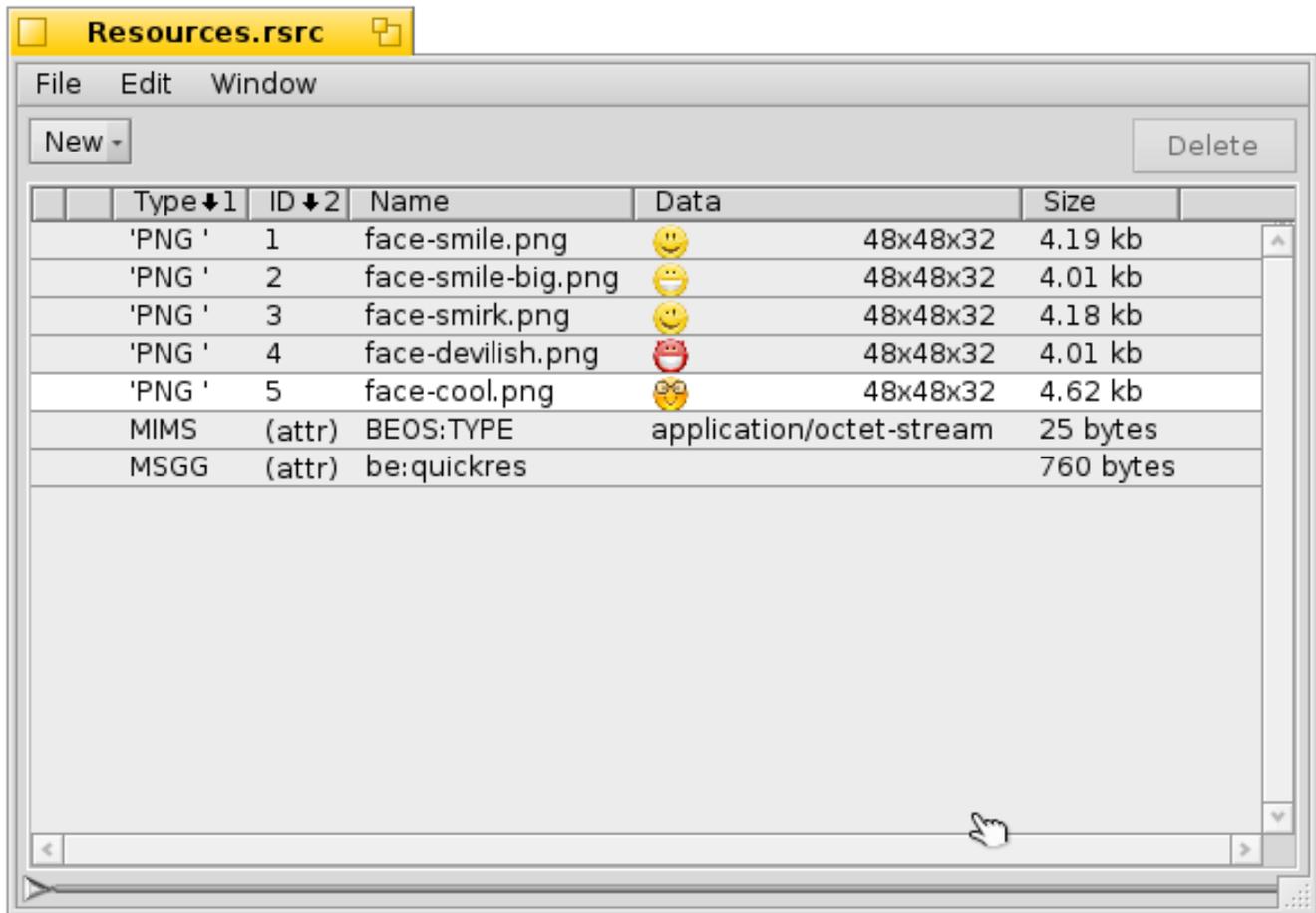
A number of tools are available for Haiku to add, delete, and edit program resources. QuickRes is the original, authoritative editor for program resources provided by Be. Unfortunately, it is not open source and not usable by GCC4-only Haiku installations. A resource editor called ResEdit is being developed for Haiku, but as of this writing, it has only been developed enough to be barely useful. rc (resource compiler) is a command-line utility which converts between text resource files and binary resource files. It is useful for making resources easily compatible with source control systems which don't deal well with binary files, such as CVS. xres is a command line utility which is used to manipulate resources. You can choose whatever tool you like, but for the purposes of these lessons, we will focus on QuickRes because it works well and eventual migration to ResEdit from QuickRes will be relatively painless when the time finally comes.

Although it's possible to directly edit a program's resources with an editor by opening the executable in the resource editor, this is not the usual – or recommended – way to add resources to an application. Resources are normally placed into a resource file that is added to the executable after it has been built. Resource files come in two formats: a binary format with a .rsrc extension and a text file format with a .rdef extension. Paladin supports both and will automatically bundle any resource files into your application if they are part of your project.

Project: Emo

Now we will be learning how to use both program resources and the Translation Kit in a little project called Emo. It loads a series of images from program resources and displays them one at a time, changing the picture whenever the user clicks on it. The sources can be found in 19Emo.zip. For the sake of space, only the parts of the code that are relevant to the lesson will be printed below.

First of all, let's start with the resource file for the project: Resources.rsrc. Creating a resource file isn't difficult, but let's look at the layout of the file when opened in QuickRes.



Each of the pictures has an entry in the file which shows its type, resource ID, name, size, and a miniature preview. Adding pictures to a resource file is just a matter of dragging them from Tracker and dropping them onto the QuickRes window. Deleting is just a couple of clicks, so nothing here is terribly complicated. The IDs are arbitrary, but they must be unique within each data type stored in the file. The name for each entry is optional, but highly recommended. You can add just about any kind of data you like to a resource file and use your own resource type codes, but try to keep them small because they have a direct impact on the executable's size. Large resources should really be kept in their own separate file. Let's move on to the code.

Surprisingly, the Haiku API does not have a view which just displays a picture, but luckily for us, making one isn't much work. First, let's look at the header, `PictureView.h`:

```

#ifndef PICTUREVIEW_H
#define PICTUREVIEW_H

#include <Message.h>
#include <Bitmap.h>
#include <String.h>
#include <View.h>

class PictureView : public BView
{
public:
    PictureView(void);
    ~PictureView(void);

    void Draw(BRect rect);
    void MouseUp(BPoint pt);

private:
    BBitmap *fBitmaps[5];
    int8 fBitmapIndex;
};

#endif

```

This doesn't appear to be anything particularly special. It's not. The class has an array of five BBitmap pointers and an index which we'll use to point to the current one. It gets only a little more complicated in the main sources for PictureView:

```

#include "PictureView.h"
#include <TranslationUtils.h>
#include <TranslatorFormats.h>

// This class is our own special control. It loads five images from the
// application's resources and places them in an array. Once loaded, it resizes
// itself to exactly fit the first bitmap. This assumption is OK since they are
// all the same size and any problems are most likely to be our fault. Most people
// don't edit program resources and if they do, let them reap the consequences of
// their actions. >:D
PictureView::PictureView(void)
: BView(BRect(0,0,100,100), "picview", B_FOLLOW_LEFT | B_FOLLOW_TOP,
    B_WILL_DRAW),
  fBitmapIndex(0)
{
    // Load up all our pictures using a loop. There are 5 different versions of
    // BTranslationUtils::GetBitmap. This is one of two which load images from
    // program resources.
    for (int8 i = 1; i <= 5; i++)
    {
        BBitmap *smiley = BTranslationUtils::GetBitmap(B_PNG_FORMAT,i);
        fBitmaps[i - 1] = (smiley && smiley->IsValid()) ? smiley : NULL;
    }

    if (fBitmaps[0] && fBitmaps[0]->IsValid())
        ResizeTo(fBitmaps[0]->Bounds().Width(),
            fBitmaps[0]->Bounds().Height());
}

```

```

PictureView::~PictureView(void)
{
}

// The BView's Draw() function is called whenever it is asked to draw itself
// on the screen. This is one of the few places where a BView's drawing commands
// can be called.
void
PictureView::Draw(BRect rect)
{
    // Alpha transparency is ignored in the default drawing mode for performance
    // reasons, so we will change the drawing mode to utilize transparency
    // information.
    SetDrawingMode(B_OP_ALPHA);

    // Set the foreground color of the BView to white
    SetHighColor(255,255,255);

    // Fill the BView's area with white. Like with most BView drawing commands,
    // the last argument is the color to use which defaults to the high color.
    // Other color choices are B_SOLID_LOW, which uses the background color, and
    // B_MIXED_COLORS, which mixes the high and low colors.
    FillRect(Bounds());

    // Draw the current bitmap on the screen
    if (fBitmaps[fBitmapIndex])
        DrawBitmap(fBitmaps[fBitmapIndex]);

    // Set the foreground color to black
    SetHighColor(0,0,0);

    // Draw a black border around the view
    StrokeRect(Bounds());
}

// Mouse handling is kinda funny. BView has three hook functions for the mouse:
//MouseDown(), which is called whenever the user presses a mouse button
// while the pointer is over the view, MouseUp, which is called whenever the user
// releases a mouse button while the pointer is over the view, and MouseMoved(),
// which is called whenever the mouse changes position while over the view. This
// gives you, the developer, a great deal of control over how your view reacts to
// any kind of mouse event.
void
PictureView::MouseUp(BPoint pt)
{
    // Go to the next image in the array or loop around to the beginning if at
    // the end.
    if (fBitmapIndex == sizeof(*fBitmaps))
        fBitmapIndex = 0;
    else
        fBitmapIndex++;

    // Force a redraw of the entire view because we've changed pictures
    Invalidate();
}

```

This class does all of the real work for the application, including handling the mouse clicks and showing the appropriate image. The code in `MainWindow.cpp` just creates a background view and a `PictureView` instance.

The highlight of this project is the call to `BTranslationUtils::GetBitmap()`. The only difference between loading a PNG file and a JPEG file here would be a different type specifier – no need to figure out how to read a JPEG file. All the heavy lifting has been done for you. Woohoo!

Going Further

There's quite a lot that you could do with what you know now. It's more a matter of getting to know the API and how to use it and less about writing C++.

- Try figuring out how to use `BView's Pulse()` hook function to automatically change bitmaps each second. Check the entry in the `BeBook` for exact details, but you'll need to set the flags sent to the `BView` constructor to `B_WILL_DRAW | B_PULSE_NEEDED` to use the `Pulse()` function.

Classes and Methods to Remember

BTranslationUtils

- `GetBitmap(const char *name, BTranslatorRoster = NULL)` – Looks for a file in the path name and, if not found, looks for a resource named name. If there is more than one with the same name, the first one is returned.
- `GetBitmap(uint32 type, int32 id, BTranslatorRoster = NULL)` – Returns the image contained in the resource identified by type and id.
- `GetBitmap(uint32 type, const char *name, BTranslatorRoster = NULL)` – Returns the image contained in the resource identified by type and name.

Learning to Program with Haiku

Lesson 20

Written by DarkWyrn



Although our focus in the last few lessons has been on the Interface Kit, another kit receives quite a lot of attention when programming in Haiku: the Storage Kit. This kit is devoted to working with files and folders on disk. We'll be doing a fast and furious crash course through it today. Don't worry, though – it's not very difficult.

Overview of the Storage Kit

The kit can be divided into six basic groups of classes.

| Class | Description |
|------------|--|
| BFilePanel | A GUI control used to navigate the filesystem and select a file or folder |
| BRefFilter | A class for filtering <code>entry_ref</code> objects. It is almost always used in combination with BFilePanel. |

| | |
|--------|--|
| BQuery | Run a search of the filesystem using attributes. Queries work only on BFS volumes. |
|--------|--|

| | |
|---------------|--|
| BVolume | This is a class which represents a disk volume, which can be a hard disk partition, removable drive, disk image, or network share. |
| BVolumeRoster | This class provides the means to get information on all volumes available on the system. |

| | |
|------------|--|
| BDirectory | BDirectory is a very useful class. It gives you ways to read the contents of a directory, create files, folders, and symbolic links, and it can find a specific entry, such as to ascertain if it exists. |
| BEntry | This is one of the most-used classes in the Storage Kit. It represents an item in a directory, such as a file or folder. A BEntry can tell you if a path exists, delete it, move it to somewhere else on the same volume, and more. |
| BEntryList | You won't probably ever use a BEntryList directly. It is a class used to define an interface for reading lists of BEntry objects that derived classes must implement. Currently BQuery and BDirectory are the only objects that do this. |
| BFile | BFile is another commonly-used class. Unsurprisingly, it represents a file on disk and provides functionality to read data from files and write to them. It provides a friendlier interface than <code>fread()</code> and friends. Like BDirectory, it is a child class of BNode, inheriting all of its attribute-related functions. |
| BNode | A BNode represents the data part of a file. The class provides functions for file locking and working with a file's attributes. |
| BNodeInfo | Although BNode can do everything BNodeInfo does, BNodeInfo provides an easy interface to common attribute tasks, such as getting a file's type. Of particular note is the static function <code>GetTrackerIcon()</code> , which is used to get the icon that would be shown in Tracker. |
| BPath | BPath gives you simple string path manipulation functions. It doesn't do many of the fancy tricks BString does, but it does do path-specific ones, like converting relative paths to absolute ones and appending entry names. |

| Class | Description |
|-----------|---|
| BStatable | This provides a friendly interface to the information provided by the <code>stat()</code> function, including entry type (file, directory, etc.), creation time, modification time, file owner, and more. It isn't used directly, though. BStatable is a pure abstract class, intended to create an interface implemented by child classes. BNode and BEntry are both child classes of BStatable. |
| BSymLink | BSymLink is next to useless. The only useful function is <code>ReadLink()</code> , but because BSymLink is a child of BNode, it doesn't know where it is in the filesystem. The link resolution functions provided by BEntry are much more convenient. |

| | |
|--------------|---|
| BAppFileInfo | This class isn't used very often unless you're developing an IDE, file browser, or something similar. It is used for getting and setting information specific to applications, such as version number, icons, supported types, and so on. |
| BMimeType | It won't be often that you'll use this class. It's for parsing MIME strings, getting access to the File Type database, and performing related tasks. The main thing that this is used for is setting the preferred application for a file type. |
| BResources | Like BAppFileInfo, you probably won't ever use this class unless you're writing a specific kind of app, such as a resource editor. It is for programmatically loading, adding, or deleting program resources. |

| | |
|------------------|--|
| The Node Monitor | Not actually a class, the Node Monitor is a service of the Storage Kit which can notify you of changes in the filesystem, such as when a volume is mounted or unmounted, when a file is created or deleted, or when the contents of a directory changes. |
|------------------|--|

Project

Using the Interface Kit does have one drawback: if you use one part of it, you almost always have to use the whole paradigm. For example, if you want to load a BBitmap from disk, you have to have a valid BApplication, even if you don't use it. The Storage Kit doesn't have this limitation. In fact, today's project is a simple console application which is simple enough for a lesson but is coded in a style that you would find in a regular Haiku application.

This project, called ListDir, is a simple version of the bash command `ls`. Given a path specified on the command line, our program will list the contents of the directory and the size of each entry in the directory. We will display the "size" of any subdirectories by listing how many entries the subdirectory has.

```
#include <Directory.h>
#include <Entry.h>
#include <Path.h>
#include <stdio.h>
#include <String.h>
```

```

// It's better to use constant global integers instead of #defines because
// constants provide strong typing and don't lead to weird errors like #defines
// can.
const uint16 BYTES_PER_KB = 1024;
const uint32 BYTES_PER_MB = 1048576;
const uint64 BYTES_PER_GB = 1099511627776ULL;

int      ListDirectory(const entry_ref &dirRef);
BString  MakeSizeString(const uint64 &size);

int
main(int argc, char **argv)
{
    // We want to require one argument in addition to the program name when
    // invoked from the command line.
    if (argc != 2)
    {
        printf("Usage: listdir <path>\n");
        return 0;
    }

    // Here we'll do some sanity checks to make sure that the path we were given
    // actually exists and it's not a file.

    BEntry entry(argv[1]);
    if (!entry.Exists())
    {
        printf("%s does not exist\n",argv[1]);
        return 1;
    }

    if (!entry.IsDirectory())
    {
        printf("%s is not a directory\n",argv[1]);
        return 1;
    }

    // An entry_ref is a typedef'ed structure which points to a file, directory,
    // or symlink on disk. The entry must actually exist, but unlike a BFile or
    // BEntry, it doesn't use up a file handle.
    entry_ref ref;
    entry.GetRef(&ref);
    return ListDirectory(ref);
}

int
ListDirectory(const entry_ref &dirRef)
{
    // This function does all the real work of the program

    BDirectory dir(&dirRef);
    if (dir.InitCheck() != B_OK)
    {
        printf("Couldn't read directory %s\n",dirRef.name);
        return 1;
    }
}

```

```

// First thing we'll do is quickly scan the directory to find the length of
// the longest entry name. This makes it possible to left justify the file
// sizes
int32 entryCount = 0;
uint32 maxChars = 0;
entry_ref ref;

// Calling Rewind() moves the BDirectory's index to the beginning of the
// list.
dir.Rewind();

// GetNextRef() will return B_ERROR when the BDirectory has gotten to the
// end of its list of entries.
while (dir.GetNextRef(&ref) == B_OK)
{
    if (ref.name)
        maxChars = MAX(maxChars, strlen(ref.name));
}
maxChars++;
char padding[maxChars];

BEntry entry;
dir.Rewind();

// Here we'll call GetNextEntry() instead of GetNextRef() because a BEntry
// will enable us to get certain information about each entry, such as the
// entry's size. Also, because it inherits from BStatable, we can
// differentiate between directories and files with just one function call.
while (dir.GetNextEntry(&entry) == B_OK)
{
    char name[B_FILE_NAME_LENGTH];
    entry.GetName(name);

    BString formatString;
    formatString << "%s";

    unsigned int length = strlen(name);
    if (length < maxChars)
    {
        uint32 padLength = maxChars - length;
        memset(padding, ' ', padLength);
        padding[padLength - 1] = '\\0';
        formatString << padding;
    }

    if (entry.IsDirectory())
    {
        // We'll display the "size" of a directory by listing how many
        // entries it contains
        BDirectory subdir(&entry);
        formatString << "\\t" << subdir.CountEntries() << " items";
    }
    else
    {
        off_t fileSize;
        entry.GetSize(&fileSize);
        formatString << "\\t" << MakeSizeString(fileSize);
    }
}

```

```

    }
    formatString << "\n";
    printf(formatString.String(),name);
    entryCount++;
}
printf("%ld entries\n",entryCount);
return 0;
}

BString
MakeSizeString(const uint64 &size)
{
    // This function just converts the raw byte counts provided by BEntry's
    // GetSize() method into something more people-friendly.
    BString sizeString;
    if (size < BYTES_PER_KB)
        sizeString << size << " bytes";
    else if (size < BYTES_PER_MB)
        sizeString << (float(size) / float(BYTES_PER_KB)) << " KB";
    else if (size < BYTES_PER_GB)
        sizeString << (float(size) / float(BYTES_PER_MB)) << " MB";
    else
        sizeString << (float(size) / float(BYTES_PER_GB)) << " GB";
    return sizeString;
}

```

Going Further

With what we now know about the Interface and Storage Kits, there is a lot that is possible. You might want to go back and expand on a previous project and see what you can do with it. If you haven't tried starting a project of your own yet, this might be a time to seriously consider one. If not, very soon we will spend more than one lesson on a project the size of which you might see in the real world to tie in everything we've been learning.

Learning to Program with Haiku

Lesson 21

Written by DarkWyrn



All of the projects that we have been working on have been small ones which didn't take very much time. Depending on the complexity, real world programming projects can take months in development or more. Today we will start working on a project which is small in comparison to many Haiku programs out there but will take us more than one lesson to complete.

Project Overview

Our project will be a relatively simple one. For about as long as UNIX has been around there has been a command called `fortune` which displays a witty saying of some sort. Often it would be run when the user logged in. We will be writing a fortune program which displays a window instead of printing to the Terminal. It will also improve upon the original because we will not limit it to one fortune file. Instead, it will randomly choose a file in the fortunes directory and choose a random entry from the file. We will also make it easy for the user to get more than one fortune if he wishes.

We will organize our code into two main parts: the GUI and the code which gets a fortune. The fortune code will be one class which is responsible for choosing the fortune file from the fortune directory, getting a random entry from the file, and returning it as a string. The GUI will consist of a multiple line text box for the fortune, a button to close the program, another to get another fortune, and an About button for the purposes of showing who wrote the program and its version. This kind of separation of code based on task is a good practice which makes for reusable code.

BFile: Make Files Sit Up and Bark

Well, maybe we can't make files act like dogs, but the BFile class is an extremely useful one which gives us one place where we can do just about anything with a file: create files and read, write, append, or even erase data, and because it inherits from BNode, we can also add, remove, or change attributes. For those not familiar, attributes are a BeOS specialty: data about a file that is not part of the file's data, but that is for another lesson. Here is a list of the most commonly-used methods available to us via BFile:

| Method | Description |
|--|---|
| <code>status_t GetSize(off_t *size) const;</code> | Gets the size of the file and places the value in <code>size</code> . |
| <code>ssize_t Read(void *buffer, size_t size);</code> | Attempts to read <code>size</code> bytes into <code>buffer</code> . The actual number of bytes read is returned. |
| <code>ssize_t Write(const void *buffer, size_t size);</code> | Attempts to write <code>size</code> bytes into <code>buffer</code> . The actual number of bytes written is returned. |
| <code>off_t Seek(off_t offset, int32 seekMode);</code> | Moves the file read/write pointer to a different location in the file. <code>seekMode</code> can be <code>SEEK_SET</code> , <code>SEEK_CUR</code> , or <code>SEEK_END</code> . These seek modes move the pointer relative to the beginning of the file, the current position, or the end of the file, respectively. |
| <code>off_t Position() const;</code> | Returns the location of the read/write pointer relative to the beginning of the file. |

| Method | Description |
|---|--|
| <code>status_t SetTo(const char *path, uint32 openMode);</code> | Set the BFile object to the path specified in path. There are other versions of this method, as well. See below for more information on this method. |
| <code>void Unset();</code> | Empties the BFile object, releasing any file handles used. This method is automatically called when the BFile object is destroyed. |
| <code>status_t InitCheck() const;</code> | Returns the current error status of the BFile. |

Setting a BFile to a location involves a little more than just giving it a file path. There are actually four different versions of `SetTo()`. They take an open mode and either a string containing the path, a `BEntry`, an `entry_ref`, or a `BDirectory` with a relative path in a string. The open mode is a combination of the following flags which includes a read or write mode:

| Mode Flag | Description |
|-------------------------------|---|
| <code>B_READ_ONLY</code> | The BFile can read from, but not write to, the file. |
| <code>B_WRITE_ONLY</code> | The BFile can write to, but not read from, the file. |
| <code>B_READ_WRITE</code> | The BFile can read from and write to the file. |
| <code>B_CREATE_FILE</code> | Create the file if it doesn't already exist. |
| <code>B_FAIL_IF_EXISTS</code> | Insist on creating a new file and failing if it exists. |
| <code>B_ERASE_FILE</code> | If the file exists, erase its data and attributes. |
| <code>B_OPEN_AT_END</code> | Set the read/write pointer to the file's end. |

The constructors for BFile have the same arguments as their respective `SetTo()` calls, but they don't return any error codes. Anything can go wrong when dealing with files, so make sure you check the error status returned from `SetTo()` or call `InitCheck()` after creating a BFile. Also, there are a limited number of file handles available on the system at any given time. Each `BEntry`, `BFile`, `BNode`, or `BDirectory` uses one when set to a path, so `Unset()` or delete them when you're done with them.

Reading and Writing Files with BFile

BFile makes reading files really easy. `Read()` takes an untyped buffer of bytes to receive data from the file and the amount of data to read. When working with text data, normally you will use either a char array or, even better, a `BString`. Here's the way to read a file using BFile and BString:

```
status_t
ReadFile(const char *path)
{
    if (!path)
        return B_BAD_VALUE;

    // Set up the file to read
    BFile file("/boot/home/Desktop/MyFile.txt", B_READ_ONLY);
    if (file.InitCheck() != B_OK)
    {
        printf("Couldn't read the file\n");
    }
}
```

```

        return B_ERROR;
    }

    off_t fileSize = 0;
    file.GetSize(&fileSize);
    if (fileSize < 1)
    {
        printf("File is empty, so no data to read\n");
        return B_OK;
    }

    // Create a buffer to hold the file data
    BString fileData;

    // We can't directly pass a BString to Read(), so we'll use the BString
    // method LockBuffer() to get a pointer to its internal storage. While the
    // BString is locked, we can't use any of its methods, but we can make
    // whatever changes we want to the internal string array that it uses.
    // LockBuffer() takes an integer of the maximum size that the array will
    // be expected to be. We'll pad the number just in case so that there are
    // no unexpected crashes.
    char *buffer = fileData.LockBuffer(fileSize + 10);

    // Read() will return the number of bytes actually read, but we're going
    // to ignore the value because we're reading in the entire file.
    file.Read(buffer, fileSize);

    // Unlock the BString so we can use its methods again.
    fileData.UnlockBuffer();

    return B_OK;
}

```

Writing files is even easier. Write() has the same parameters as Read(), but instead of copying from the file to the buffer, data is copied from the buffer to the file.

```

void
WriteFile(const char *path)
{
    if (!path)
    {
        printf("NULL path sent to WriteFile\n");
        return B_BAD_VALUE;
    }

    // Create a file, if needed, and make it both readable and writable
    BFile file(path, B_READ_WRITE | B_CREATE_FILE);
    if (file.InitCheck() != B_OK)
    {
        printf("Couldn't write file &s\n", path);
        return B_ERROR;
    }

    char testString[] = "This is some file data.\nIt's not really important.\n";
    file.Write(testString, strlen(testString));
    return B_OK;
}

```

Starting Our Project: HaikuFortune

- Open Paladin and create a new project using the GUI with MainWindow template.
- Press Alt+N or choose Add New File from the Project menu and create a file called FortuneFunctions.cpp. Make sure that you check the box to also create a corresponding header file.

The first thing we're going to do is design the class which will get the fortune from the fortune directory.

```
#ifndef FORTUNEFUNCTIONS_H
#define FORTUNEFUNCTIONS_H

#include <List.h>
#include <String.h>

extern BString gFortunePath;

class FortuneAccess
{
public:
    FortuneAccess(void);
    FortuneAccess(const char *folder);
    ~FortuneAccess(void);

    status_t SetFolder(const char *folder);
    status_t GetFortune(BString &target);
    int32 CountFiles(void) const;
    status_t LastFilename(BString &target);

private:
    void ScanFolder(void);
    void MakeEmpty(void);

    BString fPath,
            fLastFile;
    BList fRefList;
};

#endif
```

There is a reason for each method that we have in this class. First, both versions of the constructor are for convenience in creating a FortuneAccess object regardless of whether or not we know the folder we want to scan when the object is instantiated. SetFolder() allows us to change folders, should we have the desire. GetFortune() is the main reason we're creating the class in the first place: a reusable object which randomly gets a fortune from a specified folder. CountFiles() tells us how many files are available. LastFilename() gives us the name of the file from which the most recent fortune came. ScanFolder() runs through a directory and compiles a list of available files that – theoretically – have fortunes in them.

MakeEmpty() is a cleanup function which deserves a little extra explanation. The list of filenames in the fortune folder that we set is kept as a collection of entry_ref objects in a BList. There are two

problems with BList: we have to `static_cast` any time we access an object it holds, and while the BList takes care of any memory allocation it does internally, any items we give to it are not destroyed when the list is freed. This means that we have to manually go through the list, get each item, and free it ourselves. It's a pain in the neck, but, unfortunately, it's all that we have at the moment. There are better solutions out there, but that's for another time. This will work well enough for our purposes right now.

Below is the skeleton code for our class along with what each function needs to do. Your job is to write the code.

```
#include "FortuneFunctions.h"

#include <Directory.h>
#include <Entry.h>
#include <File.h>
#include <OS.h>
#include <Path.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Initialize the global path to a hardcoded value just in case.
// This happens to be different under Haiku than under previous versions
// of BeOS
BString gFortunePath = "/boot/system/data/fortunes";

FortuneAccess::FortuneAccess(void)
{
}

FortuneAccess::FortuneAccess(const char *folder)
{
    SetFolder(folder);
}

FortuneAccess::~FortuneAccess(void)
{
    // Free all items in our list
}

status_t
FortuneAccess::SetFolder(const char *folder)
{
    // Make sure that folder is valid and return B_BAD_VALUE if it isn't.
    // Set the path variable, scan the folder, and return B_OK
}

status_t
FortuneAccess::GetFortune(BString &target)
{
}
```

```

// Here's the meat of this class:
// 1) Return B_NO_INIT if fPath is empty
// 2) Return B_ERROR if the ref list is empty

// 3) This line will randomly choose the index of a file in the ref list
int32 index = int32(float(rand()) / RAND_MAX * fRefList.CountItems());

// 4) Get a pointer to the randomly-selected entry_ref
// 5) Create and initialize a BFile object in read-only mode
// 6) Check to make sure that the BFile's status is B_OK
// 7) Set fLastFile to the name property of the ref we just
// 8) Get the file's size.
// 9) If the file is empty, return B_ERROR.

// 10) Create a BString to hold the data in the file
// 11) Create a char pointer that we'll use in BFile::Read.

// 12) Initialize the pointer using BString::LockBuffer, passing the file's
//     size + 10 bytes (for safety) as the size. LockBuffer temporarily gives
//     you access to the BString's internal char array. We'll need this to
//     be able to read the file's data into the BString.

// 13) Use BFile::Read() to read the entire file using our new char pointer.
// 14) Call BString::UnlockBuffer() to invalidate our char pointer and
//     allow us to use regular BString methods again.

// 15) Use a loop to manually count the number of record separators in the
//     fortune file. The separator is the string "%\n", so use a
//     combination of BString::FindFirst and offsets in a loop to count them.

// 16) Use this line to randomly choose an entry.
int32 entry = int32(float(rand()) / RAND_MAX * (entrycount - 1));

// 17) Use FindFirst again to find the starting offset of this
//     randomly-chosen entry in the file.
// 18) Call FindFirst one last time to find the offset of the next separator
//     so we know how long the fortune is.
// 19) Create a BString to hold the fortune.
// 20) Set this new BString to the String() method plus the starting offset
//     of the BString holding the file data. This will effectively chop out
//     everything that is before our fortune in the file. It should look
//     something like this:
//     BString fortune = filedata.String() + startingOffset;
// 21) Chop off everything after our fortune in the fortune BString by
//     calling its Truncate() method.
//     Hint: length = endingOffset - startingOffset + 2
// 22) Set the parameter 'target' to our fortune data and return B_OK
}

```

```
void
```

```
FortuneAccess::ScanFolder(void)
```

```
{
```

```

// Use a BDirectory for this. Make sure that it is initialized from fPath
// properly. Empty the ref list so that we're not adding to an existing
// list. Use BDirectory::GetNextEntry to get the entry for each file in the
// folder. Use the BEntry to check to make sure that the entry is a file,

```

```

        // and, assuming so, make a new entry_ref, send it to BEntry::GetRef,
        // and add it to our ref list.
    }

void
FortuneAccess::MakeEmpty(void)
{
    // Iterate through the ref list and delete each entry_ref. After doing
    // this, call BList::MakeEmpty().
}

int32
FortuneAccess::CountFiles(void) const
{
    return fRefList.CountItems();
}

status_t
FortuneAccess::LastFilename(BString &target)
{
    // Return B_NO_INIT if the path variable is empty
    // Set the target parameter to our fLastFile property and return B_OK
}

```

Writing and Testing Our Code

Because we're dealing with non-GUI code, it's probably easiest to test all of the code with a quick-and-dirty Terminal application. In your `main()` function in `App.cpp`, comment out all of the code except the return value and write code to quickly make sure that everything works correctly. Here are some development tips that should make writing it a little easier:

- Write the destructor and `MakeEmpty()` first.
- Implement `SetFolder()` next.
- `ScanFolder()` should be written now because `GetFortune()` depends on it. Test code in `main()` should just call `SetFolder()` to some path that you want to use for testing. Using `printf()` to print out what `ScanFolder()` is doing, such as the names of each ref scanned, would be good for debugging here.
- Once `ScanFolder()` has been written, start hacking on `GetFortune()`. Once again, use `printf()` to find out what's going on.
- `LastFileName()` can be implemented whenever you feel like – it's not important until we start implementing the GUI.

Once your `FortuneAccess` class is complete and debugged, you have all that you need to write a better command-line fortune program than `fortune` itself!

Going Further

We haven't even touched the GUI yet. Think of some possible ways that we could make a simple interface to show the fortune using graphical controls. We'll make the GUI next time and our project will be complete!

Learning to Program with Haiku

Lesson 22

Written by DarkWyrn



Implementing the fortune functions in the last lesson shouldn't have been too difficult. Rest assured that you will be able to finish this project even if it was harder than expected. We will be designing the graphical interface for our fortune program today.

Usability and Interface Design

One thing that developers need to be aware of is usability, which is quality of a program's suitability for a task while matching the technical level of its users. Haiku shows the roots that BeOS had in the Macintosh operating system. Although others have made notable strides in this area, Apple products have, for the most part, been shining examples of how to design products to be both stylish and easy to use. BeOS inherited these qualities without explicitly copying the Macintosh way of doing things. This also means that Haiku developers are expected to write programs that not only avoid being unnecessarily difficult but actually help the user do their work.

Usability is its own discipline and a good study of it would take far more than the scope of these lessons. Nonetheless, we should examine some basics of design to avoid major gaffes in writing our programs. A few minutes of thinking ahead can save a lot of headaches down the road for both you and your program's users.

Determine your program's main users.

The answer to this question could be developers or general Haiku users, for example. Knowing who will be using your program will guide you in figuring out your users needs and skills. Keep in mind that if your program has a general audience, like our fortune program, you will want to design your program for nontechnical users. This will enable as many people as possible to enjoy your work. When designing for nontechnical users, it is best to keep in mind someone you know who “doesn't know much about computers,” such as a grandparent, colleague, or spouse. The more effort you put into designing your app to match the expertise of your target audience, the less support you'll be asked to do later on and the greater popularity it will enjoy.

Decide the primary and secondary tasks that your program is meant to do.

Unless you're writing a game of some kind – which is just as admirable as programs to do “work” – your program will be intended to be a tool to perform one or more tasks. The influence of a feature on the overall design of the interface should be directly proportional to the frequency of its use. A program which rips MP3s from CDs should make getting album information as easy as possible. Editing the artist name, album release date, and other information which belongs in an MP3's tags should also be pretty simple. Features which may be used rarely, if ever, should be carefully considered before including them in your program. By adding features, you add complexity. This creates more code to support and maintain and also makes using your program more difficult for less skilled users in your target audience – people who are far too often ignored or marginalized by developers and power users. Your users are best thought of as reasonably intelligent, but very busy. By prioritizing the work your program will do, you will get the most return on your investment of time and also write a higher quality application.

Think of some of the ways that your program can be extra helpful.

In some ways, your program should be thought of as a butler: you tell him to go answer the door and he does a job which some consider drudgery without taking away your being in charge. Automating mindless tasks is something which computers do really well. Most people hate this kind of stuff. Even little thoughtful touches here and there add up to an overall great user experience.

Handle problems gracefully.

Few people like to be asked pointless questions. Fewer still like to have questions asked of them every few minutes. It's a lot like a four year old child who runs up to his mom – who is otherwise engaged in conversation – and proceeds to say "Mommy" every three or four seconds in an attempt to get her attention. Overusing BAlerts creates this kind of annoyance for the user. It's OK to ask the user questions, but do so in a way that is not condescending and try to keep the frequency to a minimum whenever possible. Knowing full well that Murphy's Law certainly applies to programming, try to think of things that could go wrong and handle them gracefully. Crashes are not an option, nor is losing a user's data unless there is no other way out.

Designing Our GUI

Our fortune program isn't terribly important in the grand scheme of things, but it will be something that the user sees fairly often if it is set up to run whenever the computer is booted. This is a program for anyone, so it should be as nontechnical as possible. The main thing it will do is show a fortune when it is started, but we also should make it simple to get another fortune without having to restart the program.

We will only do one thing a couple of different ways, so we should be able to get away with using buttons instead of having to create a menu. One button can be for getting another fortune. Although there is always a close button on the window's tab, we should also use a regular button so the user can close the window by pressing the Enter key. Because we want people to have a way to know its version and who wrote this fortune program, we'll make a button which shows an About window. This should be easy to write.

Writing the Code

The first thing we should do is find out where the fortunes are being kept. Under BeOS and Zeta, they were kept in /boot/beos/etc/fortunes. Haiku keeps them in a different place: /boot/system/data/fortunes. Rather than try to guess what platform we might be using, we'll use `find_directory()`.

```
status_t find_directory(directory_which which, BPath *path,  
                        bool and_create_it = false, BVolume *vol = NULL);
```

This call takes a constant – a list of which can be found in `FindDirectory.h` – and places the location that corresponds to it into `path`. Use this call whenever you need to get a common location in the filesystem, such as the home folder, the user's settings folder, the common add-ons folder, and so on. If the actual location of such a folder should change, your program won't suddenly stop working correctly.

We are going to make a global variable to hold the path for the fortunes. Using lots of global variables is a bad idea, but one in this program won't hurt anything. Add this line to `FortuneFunctions.h`:

```
extern BString gFortunePath;
```

Place this somewhere between the includes and your class code in `FortuneFunctions.cpp`.

```
BString gFortunePath = "/boot/beos/etc/fortunes";
```

The extern keyword declares gFortunePath without defining it. If FortuneFunctions.h is used by more than one file, this will prevent linker problems. We actually define and initialize it in the main sources.

Now that we've made the necessary tweaks to our fortune access class, let's turn our attention to App.cpp. We will need to do a little additional setup before we show our fortune window.

```
#include "App.h"

#include <FindDirectory.h>
#include <OS.h>
#include <Path.h>
#include <stdlib.h>

#include "FortuneFunctions.h"
#include "MainWindow.h"

App::App(void)
: BApplication("application/x-vnd.test-HaikuFortune")
{
    BPath path;

    // We have to use an #ifdef here because the fortune files under R5
    // and Zeta are in the system/etc/ directory, but in Haiku they're
    // kept in the system/data directory. We detect the platform by using
    // a compiler definition. __HAIKU__ is defined under Haiku, but not BeOS
    // R5 or Zeta. Zeta has its own __ZETA__ definition. All three have the
    // __BEOS__ definition which, in this case, isn't at all useful.
    #ifdef __HAIKU__
    find_directory(B_SYSTEM_DATA_DIRECTORY,&path);
    #else
    find_directory(B_BEOS_ETC_DIRECTORY,&path);
    #endif

    path.Append("fortunes");
    gFortunePath = path.Path();

    // If we want the rand() function to actually be pretty close to random
    // we will need to seed the random number generator with the time. If we
    // don't, we will get the same "random" numbers each time the program is
    // run. rand() isn't really random, but this makes it close enough for us.
    srand(system_time());

    MainWindow *win = new MainWindow();
    win->Show();
}

int
main(void)
{
    App *app = new App();
    app->Run();
    delete app;
    return 0;
}
```

Now let's move on to the MainWindow class.

```
#include "MainWindow.h"

#include <Alert.h>
#include <Application.h>
#include <Button.h>
#include <Screen.h>
#include <ScrollView.h>
#include <View.h>

// We only need a couple of message constants for this app
enum
{
    M_GET_ANOTHER_FORTUNE = 'gafn',
    M_ABOUT_REQUESTED = 'abrq'
};

MainWindow::MainWindow(void)
: BWindow(BRect(0,0,300,300), "HaikuFortune", B_DOCUMENT_WINDOW,
          B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE),
  fFortune(gFortunePath.String())
{
    // Create all of the controls for our window.
    BView *back = new BView(Bounds(), "background", B_FOLLOW_ALL, B_WILL_DRAW);
    back->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
    AddChild(back);

    BButton *close = new BButton(BRect(0,0,1,1), "closebutton", "Close",
                                new BMessage(B_QUIT_REQUESTED),
                                B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM);

    close->ResizeToPreferred();
    close->MoveTo(Bounds().right - 15 - close->Frame().Width(),
                 Bounds().bottom - 15 - close->Frame().Height());
    back->AddChild(close);
    close->MakeDefault(true);

    BButton *next = new BButton(BRect(0,0,1,1), "nextbutton", "Get another",
                                new BMessage(M_GET_ANOTHER_FORTUNE),
                                B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM);

    next->ResizeToPreferred();
    next->MoveTo(close->Frame().left - 15 - next->Frame().Width(),
                 Bounds().bottom - 15 - next->Frame().Height());
    back->AddChild(next);

    BButton *about = new BButton(BRect(0,0,1,1), "aboutbutton",
                                "About" B_UTF8_ELLIPSIS,
                                new BMessage(M_ABOUT_REQUESTED),
                                B_FOLLOW_LEFT | B_FOLLOW_BOTTOM);

    about->ResizeToPreferred();
    about->MoveTo(next->Frame().left - 15 - about->Frame().Width(),
                 Bounds().bottom - 15 - about->Frame().Height());
    back->AddChild(about);
}
```

```

BRect r(15,15,Bounds().right - B_V_SCROLL_BAR_WIDTH - 15,
        next->Frame().top - 15);

fTextView = new BTextView(r, "textview",
                        r.OffsetToCopy(0,0).InsetByCopy(10,10),
                        B_FOLLOW_ALL, B_WILL_DRAW | B_PULSE_NEEDED |
                        B_FRAME_EVENTS);
fTextView->MakeEditable(false);

// BScrollViews are a little weird. You can't create one without having
// created its target. It also automatically calls AddChild() to attach its
// target to itself, so all that is needed is to instantiate it and attach
// it to the window. It's not even necessary (or possible) to specify the
// size of the BScrollView because it calculates its size based on that of
// its target.
BScrollView *sv = new BScrollView("scrollview", fTextView, B_FOLLOW_ALL, 0,
                                false, true);
back->AddChild(sv);

BString fortune;
status_t status = fFortune.GetFortune(fortune);
if (status == B_OK)
{
    BString title;
    title.Prepend("Fortune: ");
    SetTitle(title.String());

    fTextView->SetText(fortune.String());
}
else
{
    fTextView->SetText("HaikuFortune had a problem getting a fortune.\n\n"
                    "Please make sure that you have installed fortune files to "
                    "the folder ");
    fTextView->Insert(gFortunePath.String());
}

// This code is for working around a problem in Zeta. BButton::MakeDefault
// doesn't do anything except change the focus. The idea is to be able to
// press Enter to close HaikuFortune and the space bar to get another
// fortune. Zeta doesn't let us do this, so we'll leave focus on the Close
// button.
if (B_BEOS_VERSION <= B_BEOS_VERSION_5)
    next->MakeFocus(true);

// Calculate a good width for the window and center it
SetSizeLimits(45 + close->Bounds().Width() + next->Bounds().Width(), 30000,
             200, 30000) ;
r = BScreen().Frame();
MoveTo((r.Width()-Bounds().Width()) / 2.0, r.Height() / 4.0);
}

void
MainWindow::MessageReceived(BMessage *msg)
{

```

```

switch (msg->what)
{
    case M_GET_ANOTHER_FORTUNE:
    {
        BString fortune;
        status_t status = fFortune.GetFortune(fortune);

        if (status == B_OK)
        {
            BString title;
            fFortune.LastFilename(title);
            title.Prepend("Fortune: ");
            SetTitle(title.String());

            fTextView->SetText(fortune.String());
        }
        else
        {
            fTextView->SetText("HaikuFortune had a problem getting a "
                               "fortune.\n\nPlease make sure that you "
                               "have installed fortune files to the "
                               "folder ");
            fTextView->Insert(gFortunePath.String());
        }
        break;
    }
    case M_ABOUT_REQUESTED:
    {
        // Using a BAlert for the About window is a common occurrence.
        // They take care of all of the layout, so all that needs done
        // is write the text that goes in the alert.
        BAlert *alert = new BAlert("HaikuFortune",
                                   "A graphical fortune program for "
                                   "Haiku.\n\n", "OK");

        alert->Go();
        break;
    }
    default:
    {
        BWindow::MessageReceived(msg);
        break;
    }
}
}

void
MainWindow::FrameResized(float w, float h)
{
    // The BWindow::FrameResized() method is called whenever the window is
    // resized. In this case, we change the size of the text rectangle in the
    // text view used for the fortune. We have to do this because it won't do it
    // itself. Lazy. :(
    BRect textrect = fTextView->TextRect();

    textrect.right = textrect.left + (w - B_V_SCROLL_BAR_WIDTH - 40);
    fTextView->SetTextRect(textrect);
}

```

Conclusion

While the code is done and you should be able to run it and get a nice fortune, we're not quite done yet. We'll finish it next time, but for now, enjoy the satisfaction of having written a good program!

Learning to Program with Haiku

Lesson 23

Written by DarkWyrn



Writing a finished program is more than just writing the code that makes it run. Our fortune program still needs an icon and some other polishing.

Tweaking the Resources

One of the tasks remaining to further polish our program is to take care of the application's resources. They are kept in a separate file and are bundled into our program after it is built. Let's start by adding a resource file to our project.

1. In Paladin, choose Add New File from the Project menu as if you were going to add a new text file.
2. Enter Resources.rsrc for the new file's name and press Enter.

A shiny new resource file has been added to your project. Double-clicking on it will open it in the default resource file editor.

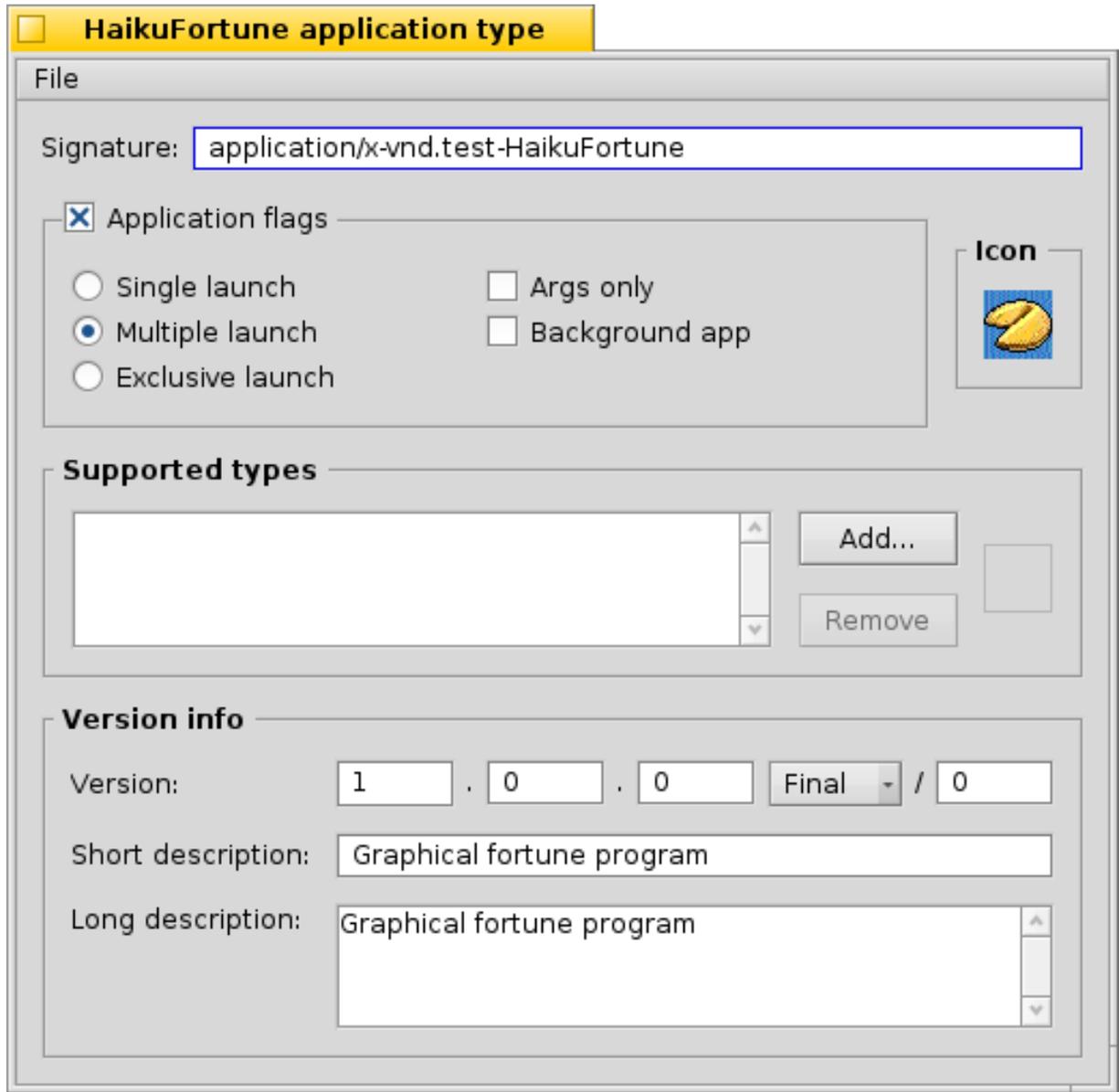
The first thing that we should do is set the program's icon, but let's address which icon we'll be setting. Haiku has three different icons that it can use: a vector icon, a large icon, and a mini icon. The last two are standard across all BeOS operating systems and are 256 color bitmaps 32 and 16 pixels square, respectively. The vector icon is a specialty format (**Haiku Vector Icon Format**) developed specifically for Haiku which looks great while taking up very little storage space. Creating HVIF icons is beyond the scope of these lessons, so we'll focus on the bitmap formats here.

1. Open Resources.rsrc in QuickRes.
2. Drag HaikuFortune16.png and HaikuFortune32.png to the QuickRes window to add them to the resource file.
3. Click on the New dropdown menu in the top left corner of the QuickRes window and choose Icon. This will create two new entries: one of ICON type and one of MICN type. These correspond to the large and mini icons.
4. Click on the entry ICON type entry once to highlight it. Wait a little bit – long enough that QuickRes won't think you're double-clicking – and click on the entry's name, which should be "New Icon". If you've done it right, a text box will appear and you can edit the name.
5. Change the ICON entry's name to `BEOS:L:STD_ICON` and press Enter to finish editing the entry's name. Capitalization matters here, by the way.
6. Do the same for the MICN entry, but name it `BEOS:M:STD_ICON`.
7. Double-click on the entry for HaikuFortune32.png to open it into an editing window.
8. Select the picture and copy it to the clipboard using the Select All and Copy commands in the Edit menu.
9. Close the window for HaikuFortune32.png and double click on the ICON entry that you just created.
10. Click on the rectangular selection tool icon and click inside the bigger grid. This will make a blue box appear around the border of the bigger grid if there isn't one already.
11. Click on Paste in the Edit menu. The picture data from HaikuFortune32.png will appear in the bigger grid.
12. Do these same steps to copy the data from HaikuFortune16.png to the smaller grid in the window.

Once you've followed these steps, save the changes to your resource file, close QuickRes, and rebuild your project. If you've done everything correctly, your program's icon will change to a nice little

fortune cookie on a blue background. If not, go back and double-check your work to find what you've missed and make the necessary changes.

We'll also need to set the version, signature, and launch flags for our application. For this, we will use the FileType Tracker add-on. Open your project's folder, select the resource file, right-click on it and choose FileType from the Add-ons submenu.



Some of the parts of this window are self-explanatory, but it's best we go over all of them anyway, starting with the application signature at the top. The signature should be set to the same string as the one passed to the `BApplication` constructor in our code minus the quotes. It's not an earth-shattering mistake to leave this out of your program's resources, but it should match the signature sent to `BApplication` if it's there.

The Application Flags group sets some options for how your program can be started. Single launch mode, which is the normal – but not default – behavior, loads one copy of our program into memory

and runs it. Attempting to run a second copy doesn't do anything except switch to the program if it hasn't been minimized. However, if the user renames the executable and tries to run it, it will run that second copy, as well. Exclusive launch prevents this and ensures that only one copy of your program will be running at one time. Multiple launch allows for multiple instances of your program to be running at the same time. If you leave out the application flags in your program's resources, this is the default behavior. Checking the Args only checkbox means that the only information given to your program will be via the command line arguments it is given. No BeOS-style messages will be sent to it. This flag is most commonly used in command line applications. The Background app checkbox allows your program to be launched in the background without showing an entry in the Deskbar. This flag is typically used for server applications like the Media Server and the Mail Daemon.

The Supported Types box is related to any file types that your program explicitly supports. For example, if you were writing a spreadsheet application, you could put in various types of spreadsheets which it was able to manipulate. Our fortune program doesn't do anything that would need this information, so we'll skip this section.

The Version Info box contains controls for the version number of our application. The dropdown box is for the quality of the release, such as alpha, beta, golden master (also known as release candidate), and final. The short description should contain a short phrase describing what your program does. The long description field should contain more information – a sentence or three – but shouldn't contain a small novel.

The icon box is for more than just looks. Right-clicking on an existing icon will allow you to remove it or edit it. An icon can be dropped onto the box to add one. By double-clicking on the icon you can edit an existing icon or add a new one, but it is edited in the Icon-O-Matic vector icon editor and not QuickRes. As a result, if you edit the icon from here, you can edit only the vector icon for your application. Once you have everything set the way you like, save your work and close the window.

Source Code Licensing

Once the resources have been set, we should give some thought to choosing a license for our source code. All source code is licensed. Which license it falls under depends on the wishes of its author. Before distributing your program, you need to consider the license under which it will be published. Doing so will determine the course and lifespan of your program. In the United States, failure to choose a license causes a restrictive set of copyright clauses to be applied to your code. If you live outside the U.S., check with the appropriate authorities in your area.

First of all, consider whether or not your source code will be available to others – closed or open source. Haiku is founded upon open source software and the community around it is largely the same way. Unless you plan on eventually charging for your program, open source is a better option. If, at some point in the future, you decide that you don't want to maintain your project any more, it will be possible for someone else to continue your efforts. Many programs on the BeBits software website are now abandoned and development cannot continue for many of these great programs because the source code was not open and the developer just disappeared.

If you do choose to make the source code for your program available, there are a plethora of open source licenses from which to choose. Here is a list of the most popular open source licenses and a quick summary of what they say:

- MIT – The code can be used pretty much however a person wants so long as the included copyright information is left intact. This is the license under which Haiku is released. It is popular with other BeOS / Haiku developers for their own projects, as well.
- GNU Public License (GPL) – Any public binary distribution of GPL software must also make the sources available. Any code which uses GPL code must also be released under the GPL. This is the most popular license for software for the Linux operating system.
- Lesser GNU Public License (LGPL) – Any public binary distribution must also make the sources available. Any code which directly uses LGPL code must also be released under the LGPL, but if a program merely links against LGPL code, this is not required. In other words, linking against an LGPL library does not require the program that links against it to have the same license.
- Mozilla Public License (MPL) – As per the GPL, but any changes to MPL-licensed code must also be submitted back to the original project.
- Public Domain – The author relinquishes all rights to the code. Literally anything can be done with the code, including removing all copyright information and anything else you could think of. Also, once code is released into the public domain, it cannot be taken back. However, nothing would stop you from using it, improving it, and re-licensing your code.

Packaging Programs for Distribution

Publishing our program is another topic that should be addressed before concluding our work. There are two methods for distribution: package files and zip archives. Both have notable benefits and drawbacks.

Packages are the more powerful format of the two. They are created using the BeOS development tool PackageBuilder. Because PackageBuilder is part of the BeOS R5 Development tools package, they are free to download, but they can't be redistributed as part of another distribution. PackageBuilder is no longer being developed and will not run in a non-hybrid GCC4 Haiku environment. The packages that it generates can be installed in such a situation, however. It can install files in multiple locations on the computer. While it does not have facilities to create links in the Deskbar, symlinks can be created and installed just like any other file in the package. As a tradeoff for more flexibility, creating a package file takes more time.

Zip archives are the simpler and faster of the two formats. Set everything up the way you want it in the application folder, run the Zip-O-Matic Tracker addon, and that's it. This, unfortunately, places more burden on the user to install the application folder and create a symlink in the Deskbar. Zip archives are recommended for simple applications which do not need files in more than one location on the hard drive.

Where to Go from Here

We have covered an enormous amount of information since Lesson 1. Amazingly, there is still a lot of ground to go if you would like to go beyond being a beginner. The best way to improve is to write code and keep learning as you go. What you know now will give you the ability to write a many different kinds of programs for Haiku and will also give you the tools to learn the rest of the Haiku API and other aspects of programming in C++. Here are some topics that you are encouraged to explore:

C++

- Exceptions
- Templates and the Standard Template Library
- Multiple Inheritance

Usability

- *The Design of Everyday Things*, Donald Norman
- *The Humane Interface*, Jef Raskin

Good Programming

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Programming for BeOS / Haiku

- *Programming the Be Operating System*, Dan Parks Sydow. This is out-of-print but is available from O'Reilly's website as a free PDF.
- The BeOS sample code projects and accompanying articles
- *The Be Book*. This is the authoritative manual on the BeOS / Haiku API.