Assignment 0: Using the Debugger

A tutorial introduction to the QT Creator debugger
written by Keith Schwarz
Hi everybody!
As part of Assignment 0, we’d like you to get a little bit of practice using the debugger in Qt Creator.
The debugger is a tool you can use to help see what your program is doing as you run it.
It’s really useful for helping find errors in your programs, and the more practice you get with it, the easier it’ll be to correct mistakes in the programs you write.
Think of this guide as a little tutorial walkthrough to help give you a sense of how to use the debugger and how to make sense of what you're seeing.
To start things off, open up the Name Hash program you ran in Part One of this assignment. Scroll down to the nameHash function so that you can see the entire function in your window.
Move your mouse cursor so that it's in the space right before the line number for line 66.

Now, click the mouse!
When you do, you should see a red circle with a little hourglass pop up.

This is called a **breakpoint**. If we run the program in debug mode, whenever the program gets to this line, it will pause and open up the debugger so we can see what’s going on.
Now, we're going to run this program in debug mode. To do so, click on the "run in debug mode" button in the bottom-right corner of the screen. It's the one just below the regular green "run" button. When you do...
... you should see something like this! Notice that a bunch of extra panels popped up in Qt Creator. We’ll talk about what each of these windows mean in a second.
In the meantime, type in the first name **Ada** and hit enter, as shown here.
Now, type in **Lovelace** as a last name, but don’t hit enter yet!
As soon as you hit enter, a bunch of things are going to pop up in Qt Creator. Don’t panic! It’s normal.
With that said, hit enter, and watch the magic happen!
Shazam! We're back in Qt Creator, and there's tons of values showing up everywhere.
There's a lot going on right here. Let's see what's happening.
First, notice that our red breakpoint now has a yellow arrow in it.
This yellow arrow indicates where in the program we are right now. The program stopped running at this line because we hit that breakpoint you set earlier.
Whenever you pop up the debugger, it’s good to figure out exactly where you are in the program that you’re running, so you’ll get into the habit of checking for this yellow arrow.
Next, let's take a look at this panel. This is called the call stack.
Right now, we know we’re in the `nameHash` function, because our helpful friend the Yellow Arrow tells us exactly what line we’re on!
However, the yellow arrow can't tell us exactly how we got to this part of the program. What part of the program actually called `nameHash`?
The call stack can tell us exactly that!
Notice that the call stack lists a series of different functions in order. Here, it has `nameHash` (where we are now) at the top, and right below that is `qMain`. 
Go and double-click the call to Main on Level 1.
When you do...
... you’ll end up over here!
Notice that the highlighted line here includes a call to the `nameHash` function. This is the part of the code that actually called `nameHash`, which is how we got to the line with the breakpoint!
Generally speaking, you can use the call stack as a way to see which function calls got us to the point where the program paused at the breakpoint!
You might notice that there's some more stuff in the call stack beyond just main and `nameHash`. What are those?
Let's find out! Double-click on the function on Level 3. (Here's what it looks like on my system; you might see something different.)
When you do, you'll see something like this. (This might be different depending on your OS. Don't panic if it doesn't exactly match.)
Yikes! This looks hairy and scary! What happened?
Whenever you start up a program in CS106B, there’s a little bit of code that we automatically call for you, which does things like setting up the console.
This code will show up in the call stack below your actual program.
You shouldn’t need to dig around this deep in the call stack, and if you do, it should probably be a message telling you to back up a bit back to code that you actually wrote.
So let’s jump back to the code that we actually wrote.
To do that, double-click on Level 0, the call to `nameHash`. When you do...
You'll be teleported back to safety!
Let's quickly recap what we've seen so far.
To set a breakpoint so that we can pause the program and look around, click in the margin just before the line number where you want to pause.
Once the breakpoint is reached, it will pull up all sorts of useful information.
The yellow arrow points out where we are right now.
The call stack shows us how we got into the current function.
Now, let's see how we can read the values of the variables in this function.
Look up at this panel over here.
This window lets you take a look at all the values of the local variables that are in scope right now.
Depending on what OS you’re using, these might be in a different order, and there might be some weird-looking ones in there in addition to nicer ones like ch and hashval.
If we ignore the weird-looking ones, we can see some nice, familiar names.
For example, here you can see the values of kLargePrime and kSmallPrime, which match the values they were declared with.

```c++
int hashVal = 0;

/* Iterate across all the characters in the first name, then the last
 * name, updating the hash at each step.
 */
for (char ch: first + last) {
    /* Convert the input character to lower case. The numeric values of
     * lower-case letters are always less than 127.
     */
    ch = tolower(ch);
    hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
}
return hashVal;
```
We can also see that, at this point, `hashVal` is still zero.
As we walk through the program one step at a time, we'll see these values change.
Now, let's take a look at this for loop:

```c++
int hashVal = 0;

/* Iterate across all the characters in the first name, then the last
 * name, updating the hash at each step.*/
for (char ch: first + last) {
    /* Convert the input character to lower case. The numeric values of
    * lower-case letters are always less than 127.*/
    ch = tolower(ch);
    hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
}
```

This hashing scheme needs two prime numbers, a large prime and a small prime. These numbers were chosen because their product is less than $2^{31} - k_{LargePrime} - 1$.

```c++
static const int kLargePrime = 16908799;
static const int kSmallPrime = 127;
```
This loop is a **range-based for loop**. It says "for each character in the string first + last, do something with that character."
Remember (from a while back) that we entered the name *Ada Lovelace*.
If we take a look at the current value of the variable ch, we can see that it has the value A. That's the first letter of the name Ada Lovelace.
So now we know where we are (line 66), how we got there (main called nameHash), and the values in the program at this point.
Now, let's do something really cool - we're going to run this program one line at a time, watching what happens at each step!
Right above the stack trace, you’ll see there are some small button icons.
These buttons let you resume the program, stop the program, walk through it one line at a time, etc.
Move your mouse so that you're hovering over the button that's third from the left. If you hover over it, it should say "step over."
Once you're confident that you're on the “Step Over” button – and not the “Step Into” or “Step Out” buttons – go and click it! When you do...
...your window should look something like this.
Okay! A few things have changed. Let’s see what’s going on.
First, notice that our helpful Yellow Arrow friend is now pointing at line 67.
We're now at the line right after the one where we stopped. You just ran a single line of the program! Pretty cool!
So what did that line of code do?
ch = tolower(ch);

This line converts ch to lower case. The tolower function takes in a character and returns a lower-case version of it, so this overwrites ch with a lower-case version of itself.
You can actually see this by looking at the values panel over on the side!

```cpp
int hashVal = 0;

/* Iterate across all the characters in the first name, then the last name, updating the hash at each step. */
for (char ch: first + last) {
    /* Convert the input character to lower case. The numeric values of lower-case letters are always less than 127. */
    ch = tolower(ch);
    hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
}
return hashVal;
```
Notice that the value associated with \texttt{ch} has changed from 'A' to 'a' – it's now in lower-case!
If you'll notice, this value is in red while all the other values are in black.
This indicates that the value here has changed since the previous step. This is a really useful way to keep track of what's changing as you run the program.
Now, let's take a look at line 67, where we are right now.
Not gonna lie, this is a pretty dense line of code. It performs some weird sort of mathematical calculation on a bunch of different values.

```cpp
int hashVal = 0;

// Iterate across all the characters in the first name, then the last * name, updating the hash at each step.
for (char ch: first + last) {
    // Convert the input character to lower case. The numeric values of * lower-case letters are always less than 127.
    ch = tolower(ch);
    hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
}

return hashVal;
```
Fundamentally, though, it's just computing some weird function of some values and stashing it into hashVal.

```cpp
int hashVal = 0;

/* Iterate across all the characters in the first name, then the last
 * name, updating the hash at each step.
 */
for (char ch: first + last) {
    /* Convert the input character to lower case. The numeric values of
    * lower-case letters are always less than 127.
    */
    ch = tolower(ch);
    hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
}
return hashVal;
```
Let's go run that line of code and see what happens!
Hover over the “Step Over” button, confirm that the button you’re clicking really is “Step Over,” and click it! When you do...
```cpp
int nameHash(string first, string last) {
    /* This hashing scheme needs two prime numbers, a large prime and a small prime. These numbers were chosen because their product is less than 2^31 - kLargePrime - 1. */
    static const int kLargePrime = 16908799;
    static const int kSmallPrime = 127;

    int hashVal = 0;
    /* Iterate across all the characters in the first name, then the last name, updating the hash at each step. */
    for (char ch : first + last) {
        /* Convert the input character to lower case. The numeric values of lower-case letters are always less than 127. */
        ch = tolower(ch);
        hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
    }
    return hashVal;
}
```

...you'll end up with something like this!
Let's see what's changed.
First, notice that the value stored in `hashVal` changed to 97. We know that it changed because the value is in red, and we know that nothing else changed because nothing else is in red!
Second, notice that we’re back up at the top of the for loop, since that’s where the yellow arrow is pointing. We ended up back here because this is the next line that gets executed.
We just single-stepped through a single iteration of that loop! Pretty cool!
Let's go do it again!
Again, move your mouse over the Step Over button (and make sure it says “Step Over” and not something else!), then click it.
Now we're here! Notice that ch now has the value 'd', which is the second letter of the name Ada.
Go click "Step Over" again to run this line of code.
You should be here now. Notice that none of the values changed. That makes sense, since all we did was convert a lower-case 'd' to a lower-case 'd'.
Now, click “Step Over” one more time.
You'll now be at this point in the program. We've covered up the value of hashVal in this image, because at this point you should be able to see what hashVal is by reading the value in the side pane. This is the special value we want you to tell us when submitting the assignment!
To finish up this section on the debugger, we’d like to show you two last little techniques that you might find useful when debugging programs.
To start this off, click on the the breakpoint that we set earlier in the program. If you do...
... it should clear the breakpoint. Now, if we were to run this program again in debug mode, it would not stop at this point, since nothing’s telling it to!
Now, take a look back at these buttons.
Hover your mouse over the one that’s on the far right. When you hover over it, it should say “Step Out.”
If you click this button, it will keep running this function up until it completes and returns.
Now, go click that button. If you did everything right...
...you should end up with something that looks like this!
Let's take a minute to get our bearings. Where exactly are we?
Well, the yellow arrow indicates that we're back in main again. Cool!
We can see that the `nameHash` function returned 1967457. Thanks, debugger!
But if you look up over here in the values window, you can see that hashValue has some really weird-looking number stored in it. (You’ll almost certainly see something different on your system.)
But it looks like we're setting `hashValue` equal to the number that was returned by the `nameHash` function. What's going on?
This is pretty cool, actually!
What's happened is that we've just returned from `nameHash` with a value, but since we're going through the program one step at a time, we haven't actually assigned that value to `hashValue` yet!
Let’s do a “Step Over” so that we can finish executing this line. Click “Step Over,” and if you did everything right...
...you should see the right value get stored (notice it's in red!) and we've moved to the next line.
At this point, we've seen just about everything we care about. Rather than single-stepping all the way to the end, let's just tell the program to keep on running.

```cpp
int hashValue = nameHash(first, last);

cout << "The hash of your name is: " << hashValue << endl;
return 0;
```

/* This is the actual function that computes the hash code. We're going to talk more about what hash functions do later in the quarter. In the meantime, think of it as a function that scrambles up the characters of the input and produces a number.

For those of you who are more mathematically inclined, this function treats each character in the input name as a number between 0 and 128. It then uses them as coefficients in a polynomial over the finite field $\mathbb{F}_p$, where $p$ is a large prime number, and evaluates that polynomial at some smaller prime number $q$. (You aren't expected to know this for CS106B,
To do this, click on this button. If you hover over it, it says “Continue,” and that button means “unpause the program and let it keep running from here.”
If you do, you should see something like this. (The program window might not automatically pop up. That’s okay! Just open it manually.) Our program is now done running!
So there you have it! You’ve now gotten more familiar with the debugger!
You know how to set a breakpoint to pause the program at a particular point.
You know how to read the call stack and to see the values of local variables.
You know how to single-step the program and see what values change.
You know how to run a function to completion, and how to let the program keep on running.
As you write more and more complicated programs this quarter, you’ll get a lot more familiar using the debugger and seeing how your programs work.
And, if you continue to build larger and larger pieces of software, you’ll find that knowing how to use a debugger is a surprisingly valuable skill!
Hope this helps, and welcome to CS106B!