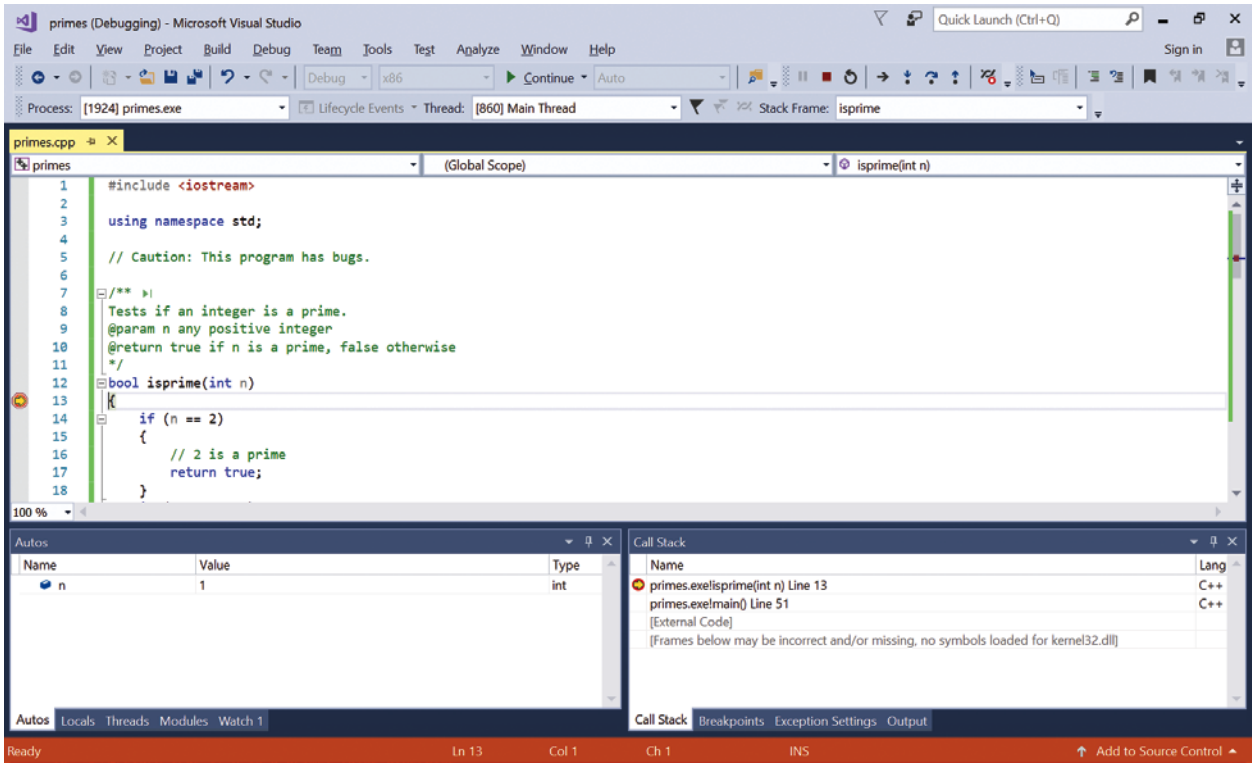
WORKED EXAMPLE 5.2

Using a Debugger

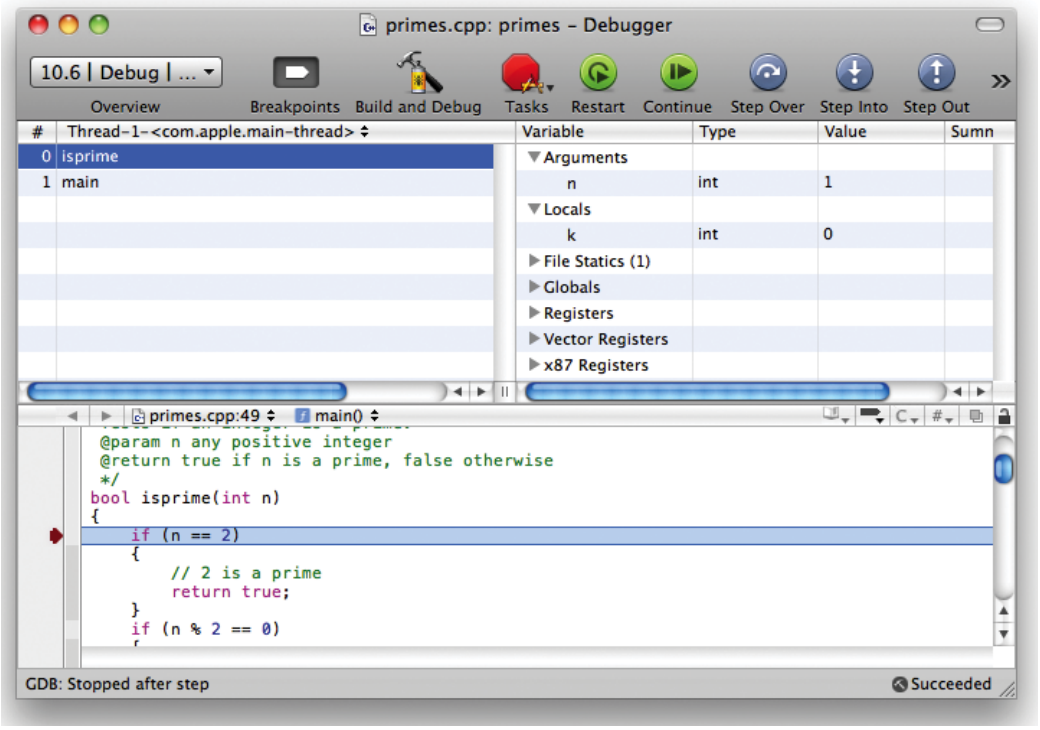
As you have undoubtedly realized by now, computer programs rarely run perfectly the first time. At times, it can be quite frustrating to find the errors, or bugs, as they are called by programmers. Of course, you can insert print statements into your code that show the program flow and values of key variables. You then run the program and try to analyze the printout. But if the printout does not clearly point to the problem, you need to add and remove print statements and run the program again. That can be a time-consuming process.

Modern development environments contain a **debugger**, a program that helps you locate bugs by letting you follow the execution of a program. You can stop and restart the program and see the contents of variables whenever the program is temporarily stopped. At each stop, you can decide how many program steps to run until the next stop.

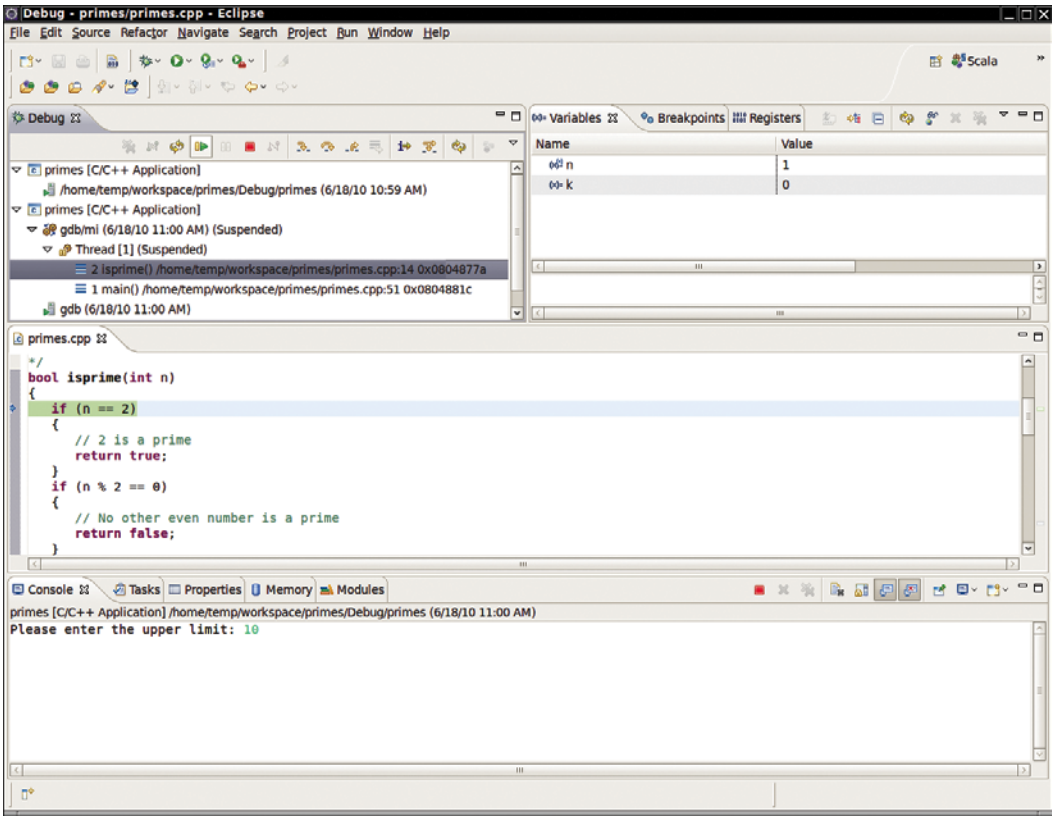
Step 1 Just like compilers, debuggers vary widely from one system to another. The debuggers of most integrated environments have a similar layout—see the examples below. You will have to find out how to prepare a program for debugging, and how to start the debugger on your system. With many development environments, you can simply pick a menu command to build your program for debugging and start the debugger.



The Visual Studio Debugger



The XCode Debugger



The Eclipse Debugger

Step 2 Once you have started the debugger, you can go a long way with just three debugging commands: "set breakpoint", "single step", and "inspect variable". The names and keystrokes or mouse clicks for these commands differ widely between debuggers, but all debuggers support these basic commands. You can find out how, either from the documentation or a lab manual, or by asking someone who has used the debugger before.

Step 3 When you start the debugger, it runs at full speed until it reaches a **breakpoint**. Then execution stops, and the breakpoint that causes the stop is displayed.

You can now inspect variables and step through the program a line at a time, or continue running the program at full speed until it reaches the next breakpoint. When the program terminates, the debugger stops as well.

Breakpoints stay active until you remove them, so you should periodically clear the breakpoints that you no longer need.

Once the program has stopped, you can look at the current values of variables. Some debuggers always show you a window with the current local variables. On other debuggers you issue a command such as "inspect variable" and type the variable name. If all variables contain what you expected, you can run the program until the next point where you want to stop.

Step 4 Running to a breakpoint gets you there speedily, but you don't know what the program did along the way. For a better understanding of the program flow, you can step through the program a line at a time. Most debuggers have two step commands, one usually called "step into", which steps inside function calls, and one called "step over", which skips over function calls. You should step into a function to check whether it carries out its job correctly. Step over a function if you know it works correctly.

Step 5 Finally, when the program has finished running, the debugging session is also finished. To run the program again, you need to start another debugging session.

A debugger can be an effective tool for finding and removing bugs in your program. However, it is no substitute for good design and careful programming. If the debugger does not find any errors, it does not mean that your program is bug-free. Testing and debugging can only show the presence of bugs, not their absence.

Sample Session

Here is a simple program for practicing the use of a debugger. The program is supposed to compute all prime numbers up to a number *n*. (An integer is defined to be prime if it is not evenly divisible by any number except by 1 and itself. Also, mathematicians find it convenient not to call 1 a prime. Thus, the first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19.)

worked_example_2/primes.cpp

```
#include <iostream>

using namespace std;

// Caution: This program has bugs.

/**
   Tests if an integer is a prime.
   @param n any positive integer
   @return true if n is a prime, false otherwise
*/
bool isprime(int n)
{
```

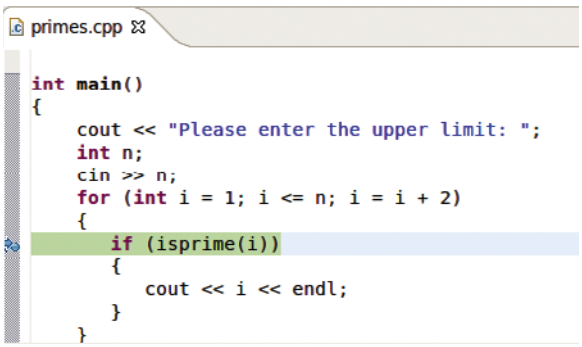
```
14    if (n == 2)
15    {
16       // 2 is a prime
17       return true;
18    }
19    if (n % 2 == 0)
20    {
21       // No other even number is a prime
22       return false;
23    }
24
25    // Try finding a number that divides n
26
27    int k = 3; // No need to divide by 2 since n is odd
28    // Only need to try divisors up to sqrt(n)
29    while (k * k < n)
30    {
31       if (n % k == 0)
32       {
33          // n is not a prime since it is divisible by k
34          return false;
35       }
36       // Try next odd number
37       k = k + 2;
38    }
39
40    // No divisor found. Therefore, n is a prime
41    return true;
42 }
43
44 int main()
45 {
46    cout << "Please enter the upper limit: ";
47    int n;
48    cin >> n;
49    for (int i = 1; i <= n; i = i + 2)
50    {
51       if (isprime(i))
52       {
53          cout << i << endl;
54       }
55     }
56     return 0;
57 }
```

When you run this program with an input of 10, then the output is

```
1
3
5
7
9
```

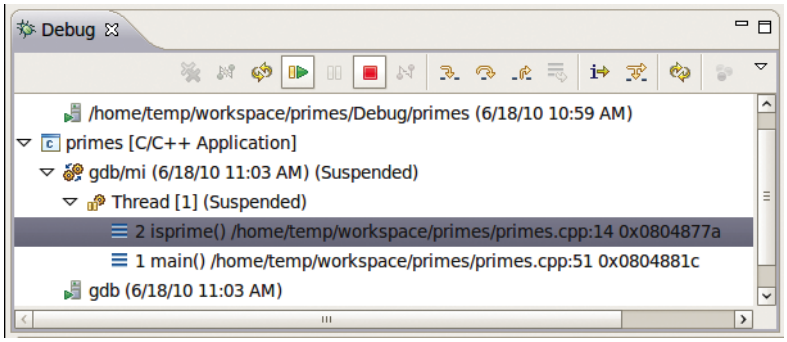
That is not very promising. It looks as if the program just prints all odd numbers. Let us find out what it does wrong by using the debugger.

First, set a breakpoint in line 51 and start debugging the program. On the way, the program will stop to input a value into `n`. Type 10 at the input prompt. The program will then stop at the breakpoint.

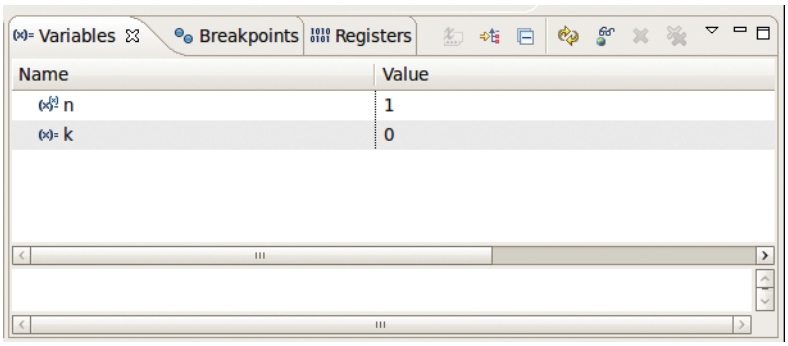


Now we wonder why the program treats 1 as a prime. Step into the `isprime` function.

Note the call stack display. It shows that the `isprime` function is currently active, and it is called by the `main` function.



Inspect the variable `n` to confirm that it is currently 1.



Execute the "step over" command a few times. You will notice that the program skips the two `if` statements. That's not surprising—1 is an odd number. Then the program skips over the `while` statement and is ready to return `true`, indicating that 1 is a prime.

Inspect the value of `k`. It is 3, which explains why the `while` loop was never entered. It looks like the `isprime` function needs to be rewritten to treat 1 as a special case.

Next, we would like to know why the program doesn't print 2 as a prime even though the `isprime` function recognizes that 2 is a prime. Continue the debugger. It will stop at the breakpoint in line 51.

Note that `i` is 3. Now it becomes clear. The `for` loop in the `main` function only tests odd numbers. Either `main` should test both odd and even numbers, or better, it should just handle 2 as a special case.

Finally, we would like to find out why the program believes 9 is a prime. Continue debugging until the breakpoint is hit with `i = 9`. Step into the `isprime` function. Now use "step over" repeatedly. The two `if` statements are skipped, which is correct since 9 is an odd number. The program again skips past the `while` loop. Inspect `k` to find out why. `k` is 3. Look at the condition in the `while` loop. It tests whether `k * k < n`. Now `k * k` is 9 and `n` is also 9, so the test fails.

When checking whether `n` is prime, it makes sense to only test divisors up to $\sqrt{n}$. If n can be factored as $p \times q$, *then the factors can't both be greater than* $\sqrt{n}$. But actually that isn't quite true. If n is a perfect square of a prime, then its sole nontrivial divisor is equal to $\sqrt{n}$. That is exactly the case for $9 = 3 \times 3$. We should have tested for `k * k <= n`.

By running the debugger, we discovered three bugs in the program:

- `isprime` falsely claims 1 to be a prime.
- `main` doesn't test 2.
- There is an off-by-one error in `isprime`. The condition of the `while` statement should be `k * k <= n`.

Here is the improved program:

```
#include <iostream>

using namespace std;

/**
   Tests if an integer is a prime.
   @param n any positive integer
   @return true if n is a prime, false otherwise
*/
bool isprime(int n)
{
   if (n == 1)
   {
      // 1 is not a prime
      return false;
   }
   if (n == 2)
   {
      // 2 is a prime
      return true;
   }
   if (n % 2 == 0)
   {
      // No other even number is a prime
      return false;
   }

   // Try finding a number that divides n

   int k = 3; // No need to divide by 2 since n is odd
   // Only need to try divisors up to sqrt(n)
   while (k * k <= n)
   {
      if (n % k == 0)
      {
         // n is not a prime since it is divisible by k
         return false;
      }
      // Try next odd number
      k = k + 2;
   }
```

```
42
43    // No divisor found. Therefore, n is a prime
44    return true;
45 }
46
47 int main()
48 {
49    cout << "Please enter the upper limit: ";
50    int n;
51    cin >> n;
52    if (n >= 2)
53    {
54       cout << 2 << endl;
55    }
56    for (int i = 3; i <= n; i = i + 2)
57    {
58       if (isprime(i))
59       {
60          cout << i << endl;
61       }
62    }
63    return 0;
64 }
```

Is our program now free from bugs? That is not a question the debugger can answer. Remember, testing can only show the presence of bugs, not their absence.