Debugging

Some have said that any monkey can write a program – the hard part is debugging it. While this is somewhat oversimplifying the difficult process of writing a program, it is sometimes more time consuming and frustrating to debug a program than it was to write it in the first place. However there are tools to help you! The purpose of this lecture is to introduce you to some of these tools.

In general we have two methods of debugging:

1. Adding print statements, which has the benefit of working in any programming language and in any environment, but is very tedious and can sometimes actually lead to errors
2. Using the debugging utilities in NetBeans or DrJava or Visual Studio

There is also a debugger in Unix called **jdb** for the Java environment and **gdb** for C/C++. We won't cover it here, but you should be aware that it exists if you plan to do more development under Unix.

Debugging with Print Statements

You have probably already used this method for debugging. The basic idea is very simple. If your program is not working correctly, try inserting print statements (in C++ this would be `cout` statements) to narrow down where the error is. The print statements could be used to locate the section of code that contains the problem, and they might output the values of some variables that would aid in debugging.

As one example, consider the code below:

```
int n = 10;
int sum = 10;
while (n > 1)
{
    sum = sum + n;
    n--;
}
cout << "The sum of the integers 1 to 10 is " << sum << endl;
```

Upon running this program, it doesn't add the numbers from 1-10. You can probably eyeball what is wrong but if you don't see it then to find out what is wrong, you can trace the variables `n` and `sum` by inserting output statements as follows:

```
int n = 10;
int sum = 10;
while (n > 1)
{
```

```
    cout << "At the beginning of the loop: n = " << n << endl;
    cout <<
              "At the beginning of the loop: sum = " << sum << endl;
   sum = sum + n;
   n--;
   cout << "At the end of the loop: n = " << n << endl;
   cout << "At the end of the loop: sum = " << sum << endl;
}
cout << "The sum of the integers 1 to 10 is " << sum << endl;
```

The first four lines of the execution are as follows:

```
At the beginning of the loop: n = 10
At the beginning of the loop: sum = 10
At the end of the loop: n = 9
At the end of the loop: sum = 20
```

We can immediately see that something is wrong. The variable `sum` has been set to 20. Since it was initialized to 10, it is set to 10 + 10, which is incorrect if we want to sum the numbers from 1 to 10.

Let's illustrate both good and bad debugging techniques with an example. Suppose our program is presenting a menu where the user can select 'A' or 'B'. The purpose of the following code is to validate user input from the keyboard and to make the user type a choice again if something other than 'A' or 'B' is entered. To be more user-friendly, the program should allow users to make their selections in either uppercase or lowercase.

```
#include <iostream>
using namespace std;

int main()
{
  char c = ' ';
  do
  {
    cout << "Enter 'A' for option A or 'B' for option B." << endl;
    cin.get(c);
    tolower(c);
  } while ((c != 'a') || (c != 'b'));
  return 0;
}
```

The program runs but if we enter any letter we get the menu twice!

```
Enter 'A' for option A or 'B' for option B.
f
```

```
Enter 'A' for option A or 'B' for option B.
Enter 'A' for option A or 'B' for option B.
```

If we employ the "guessing" technique we might try to input the character twice. After all, the prompt is showing up twice!

```
cin.get(c);
cin.get(c);
```

Such a change will "fix" this error, but it may cause other problems. In this case, we're still stuck in the loop. We can add a print statement to see if we can figure out what is going wrong:

```
cin.get(c);
cin.get(c);
cout << "c is " << c << endl;
```

The output becomes:

```
Enter 'A' for option A or 'B' for option B.
f
c is
```

This should jog our memory! c is blank. This is because it read in the newline after the character is entered. We can use the ignore function to discard it:

```
do
{
  cout << "Enter 'A' for option A or 'B' for option B." << endl;
  cin.get(c);
  cin.ignore(); // Ignore newline after character
  tolower(c);
} while ((c != 'a') || (c != 'b'));
```

At this point we have corrected the syntax error and our program will compile, but it will still not run correctly. A sample execution is shown below:

```
Enter 'A' for option A or 'B' for option B.
C
Enter 'A' for option A or 'B' for option B.
B
Enter 'A' for option A or 'B' for option B.
A
Enter 'A' for option A or 'B' for option B.
(Control-C)
```

The program is stuck in an infinite loop even when we type in a valid choice. The only way to stop it is to break out of the program (in the sample output by hitting Control-C, but you may have to use a different method depending on your computing environment).

At this point we could employ tracing to try to locate the source of the error. Here is the code with output statements inserted:

```
do
{
  cout << "Enter 'A' for option A or 'B' for option B." << endl;
  cin.get(c);
  cin.ignore(); // Ignore newline after character
  tolower(c);
  cout << "c is: " << c << endl;
} while ((c != 'a') || (c != 'b'));
```

Sample output is as follows:

```
Enter 'A' for option A or 'B' for option B.
A
c is: A
Enter 'A' for option A or 'B' for option B.
```

The `cout` statement makes it clear what is wrong—the string `s` does not change to lowercase. A review of the `tolower( )` documentation reveals that this function does not change the calling char, but instead returns a new char converted to lowercase. The calling char remains unchanged. To fix the error, we can assign the lowercase char back to the original with

```
c = tolower(c);
```

However, we're not done yet. Even after fixing the lowercase error, the program is still stuck in an infinite loop, even when we enter `'A'` or `'B'`. A novice programmer might "patch" the program like so to exit the loop:

```
do
{
   cout << "Enter 'A' for option A or 'B' for option B." << endl;
   cin.get(c);
   cin.ignore(); // Ignore newline after character
   c = tolower(c);
   if ( c == 'a')
    break;
   if (c == 'b')
    break;
}
while ((c != 'a') || (c != 'b'));
```

This forces the loop to exit if `'a'` or `'b'` is entered, and it will make the program work. Unfortunately, the result is a coding atrocity that should be avoided at all costs. This "quick fix" does not address the root cause of the error—only the symptoms. Moreover, such patches usually won't work for new cases. This particular fix also results in inconsistent code because the expression `((c! = 'a') || (c! = 'b'))` becomes

meaningless when we already handle the 'a' and 'b' with the `if` and `break` statements.

To really find the bug, we can turn again to tracing, this time focusing on the Boolean values that control the `do-while` loop:

```
do
{
  cout << "Enter 'A' for option A or 'B' for option B." << endl;
  cin.get(c);
  cin.ignore(); // Ignore newline after character
  c = tolower(c);
   cout << "c != 'a' is " << (c!= 'a') << endl;
   cout << "c != 'b' is " << (c!= 'b') << endl;
   cout << "(c != 'a') || (c != 'b')) is "
          << ((c != 'a') || (c != 'b')) << endl;
}
while ((c != 'a') || (c != 'b'));
```

The sample output is now as follows:

```
Enter 'A' for option A or 'B' for option B.
A
c != 'a' is 0
c != 'b' is 1
(c != 'a') || (c != 'b')) is 1
```

Since `c` equals 'a', the statement `(c != 'a')` evaluates to `false` and the statement `(c != 'b')` evaluates to `true`. When combined, `(false || true)` is `true`, which makes the loop repeat. In spoken English it sounds like "c not equal to 'a' or "c not equal to 'b'" is a correct condition to repeat the loop. After all, if the character typed in is not 'a' or the character typed in is not 'b', then the user should be prompted to try again. Logically however, if `(c != 'a')` is `false` (i.e., the character is `'a'`), then `(c != 'b')` must be true. A character cannot make both expressions `false`, so the final Boolean condition will always be `true`. The solution is to replace the "or" with an "and" so that the loop repeats only if `(c != 'a') && (c != 'b'))`. This makes the loop repeat as long as the input character is not 'a' and it is not 'b'.

An even better solution is to declare a `boolean` variable to control the `do-while` loop. Inside the body of the loop we can set this variable to `false` when the loop should exit. This technique has the benefit of making the code logic easier to follow, especially if we pick a meaningful name for the variable. In the following example it is easy to see that the loop repeats if `invalidKey` is `true`:

```
bool invalidKey = true;
do
{
  cout << "Enter 'A' for option A or 'B' for option B." << endl;
  cin.get(c);
  cin.ignore(); // Ignore newline after character
  c = tolower(c);
```

```
    if (c == 'a')
        invalidKey = false;
    else if (c == 'b')
        invalidKey = false;
    else
        invalidKey = true;
} while (invalidKey);
```

The approach of adding print statements has the nice benefit that it works in any programming environment. As such, it is useful to know. However, there are definite drawbacks. First, it is very tedious. The programmer must first figure out where to add the print statements, and then add them and recompile the program. After running the program, more print statements may likely be needed, so they will need to be added and recompiled again. Finally, when the program works as intended, the programmer must go back and remove all of those print statements that were added. Sometimes the process of adding and removing print statements can mess up the program if the programmer deletes an actual line of text by mistake, or if the programmer adds new temporary variables or logic to aid in debugging!

There must be a better way, and there is: the solution is to use an integrated debugger.

Debugging in Visual Studio

We'll give a short demo of the same program in class.