P2T: C Programming under Linux

Lab 2: Flow of Control and Loops

Introduction

The purpose of this laboratory is to help you become familiar with the concepts of conditional branching and loops in a C program. This material has been covered in Lecture 3 of the C component of P2T: C Programming under Linux. In this lab `if-else` and `switch` conditional statements and `do-while` and `for` flow control statements are covered.

Handling User Input

Before we begin the main body of the labs, we'll take a short digression to discuss how to get input from outside your program (as a partner to the end of Lab 1, where we showed how to display output from your code).

The variety and complexity of programs that can be written increases immensely as soon as we are able to accept input from a user. To understand the technicalities of getting user input requires more C knowledge than you have at this point, but we shall present a recipe that can be used, in order that you can write more interesting programs. Don't worry if you don't understand it all at the moment, we'll be covering the important points in the next few lectures.

Below is a simple program that will read some input that the user types and attempt to read a single float, character and integer from the command line.

```
1   #include <stdio.h>
2
3   int main (void) {
4     char op, buffer[100];
5     float x;
6       int y;
7
8     printf("Enter float , character and int\n");
9     fgets(buffer,sizeof(buffer),stdin);
10    sscanf(buffer, "%f %c %d", &x, &op, &y);
```

```
11
12      printf("This is the contents of x: %.2f\n", x);
13      printf("This is the contents of op: %c\n", op);
14      printf("This is the contents of y: %d\n", y);
15  }
```

If you were to compile and run this small program you would see something like this at the shell prompt:

```
Enter float , character and int
```

If we were to add some input we'd see the following displayed on the screen.

```
Enter float , character and int
1.0 + 33
This is the contents of x: 1.00
This is the contents of op: +
This is the contents of y: 33
```

So, this program reads in a floating point number, a character, and an integer, and stores them in individual variables. How does it do that?

On line 4 of the program we declare a variable called `op`, which is a single character, and something called `buffer[100]`. This syntax denotes a thing called an "array", which will be covered in the next lectures. For now, you can think of it as a buffer which can hold 100 `char`s in a row. On line 5 and 6 we declare more variables to store a float and an integer. We ask the user to provide some input on line 8 and then we actually read in some input on line 9 using the `fgets()` function.

The general form of the `fgets()` function is: `fgets(name, size, stdin);`

- name - is the name of the character array. The line, including the end-of-line character, is read into this array;
- size - the maximum number of characters to be read. `fgets()` reads until it gets a line complete with ending `'\n'` or it reads `size - 1`;
- stdin - specifies that we want to read our input from the STDIN of the process (which will often be the keyboard of the user);

In this case we use the `sizeof()` function so you don't have to figure out what to enter instead of "size" in the `fgets()` function. Whatever you type at the keyboard will then be contained in the variable "name", *including* the return character.

Once a line of text has been read from the input, the values contained in it can be extracted into variables. In this case the `sscanf()` function re-reads the string and breaks it into parts. The general format of a `sscanf()` statement is: `sscanf(name, format, &variable1, ...)`

- name - is the name of the character array to read from;
- format - is a format string similar that used in `printf` to describe the type of each piece of data to be read (with a few niggle differences);
- &variable - is a sequence of variables to store the values once the've been read from the buffer, with `&` appended to their names. The `&` lets the sscanf function modify the contents of the variables, for reasons explained in a later lecture. As with `printf`, you need one variable for every % style "conversion specifier" in the format string, and the variables should match the types specified.

Once the values have been read from the input and stored in variables they can be used like any other variable in our C program. In our example above we simply write them back out to the screen but we would, of course, generally do something less trivial with them.

Conditional Branching (if-else and switch)

Conditional Branching is about controlling the flow-of-execution of a program or, equivalently, the conditional evaluation of statements. That is, choosing to follow one, or another, set of instructions dependant on the result of a test. For example, bank account security allows withdrawal of money only if a pin has been verified (*test 1*) and there is enough money in an account to cover the withdrawal (*test 2*). C provides two structures for this kind of decision making, via the `if` and `switch` conditional statements. These statements both evaluate tests based on C's representation of "false" (`0`) and "true" (`>0`) to make a decision on whether or not to take an action.

The simplest decision making statement in C is the `if` statement, which can be used in a number of different ways:

- A simple `if` statement, which executes code only if a condition is met.

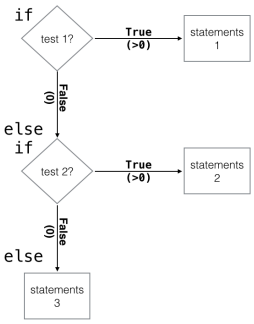
```
if ( condition is true ) {
  execute statements here;
}
```

- An `if-else` statement, which executes statement1 if a condition is met or statement2 otherwise.

```
if ( condition is true ) {
  statement1;
}
else {  //otherwise the condition is false
  statement2;
}
```

- An `if-else-if-else` statement, which executes statement1 if a condition1 is met or statement2 if condition 2 is subsequently met or statement3 if neither condition 1 or 2 is met.

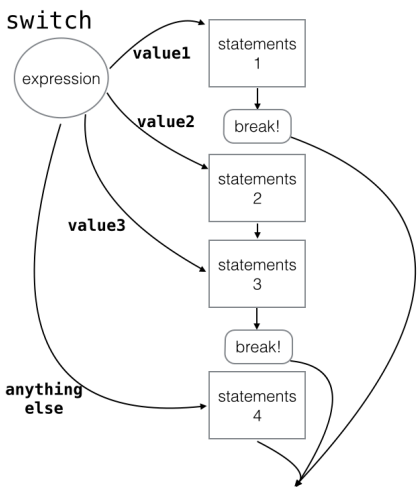
```
if (  condition 1 ) {
  statement1;
}
else if ( condition 2 ) {
  statement2;
}
else {
  statement3;
}
```



Multiple `else if` blocks can be included between the starting `if` block and the final `else`, each with their own alternative test. In this case, the different tests are made from top to bottom, with the first test to pass being the one which is executed. The `else` at the end is only used if all of the alternative conditions before it are `false`.

`Switch` statements can be used to represent "multiple option" choices (these types of problem could also be solved by a large `if-else-if-else` statement but they soon become unwieldy and difficult to read). The *expression* in the switch statement below is evaluated to return an integer value, which is then compared to the values in each case below (from top to bottom). At the first place it matches that block of code is executed. If nothing matches, the default block is executed (the default case is strictly optional, but omitting it is not safe unless you have cases for all possible values of the expression). The general form of `switch` statement is:

```
switch (expression) {
  case value1:
    statements1;
    break;
  case value2:
    statements2;
  case value3:
    statements3;
    break;
  default:
    statements4;
  }
```



Here, the "break" statement is used to jump to the outside of the switch block. Notice that the case statements aren't within their own blocks, so if expression was equal to value2, the lack of a break between the two cases would lead to statements2 *a*nd statements3 being executed. You can use this to save rewriting code, but you also need to be mindful of the need for break statements if you want the cases to be independent.

`switch`...`case` constructs only switch on integer values, meaning that usually you want the switch expression to perform a calculation with a limited number of results. However, as `char` variables are actually numeric (being an 8-bit number representing the index of the particular symbol they hold), you can also use `char`s and `char` literals in these constructs.

Challenge Questions (1)

The Challenge Questions in this Lab are broken into two sections. You should complete both this question, and the question at the end of the lab. As mentioned in Lab 1, marks are given for code which compiles and executes (even if it doesn't work perfectly), good layout and style, and good comments, as well as for a good solution to the actual problem.

Exercise 1: Calculator

Write a simple calculator program that can perform the 4 basic arithmetic operations (addition, subtraction, multiplication, division). Specifically: when run,

the program should display a text prompt asking for input (two numbers and an operation) - you may take these separately, or as a single line, as long as your code appropriately asks. The program should perform the correct operation on the numbers, and should then print the result to the screen, and exit.

Loops (while and for)

We often encounter the situation when programming that we want to repeat the same set of actions multiple times. For example: if we wanted to print the numbers between 1 and 5 in sequence, we *could* write a program that has 5 explicit `printf` statements:

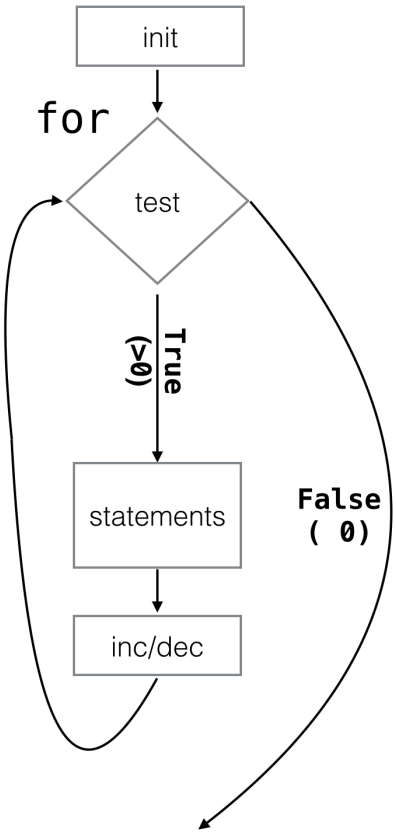
```
#include <stdio.h>

int main(void) {
  printf("%d\n", 1);
  printf("%d\n", 2);
  printf("%d\n", 3);
  printf("%d\n", 4);
  printf("%d\n", 5);
}
```

Modifying this code to print the numbers 1 to 100 would require us to to copy the `printf` statement an additional 94 times, changing the number each time. Loop constructs give us a much better way to accomplish the same task, by simply repeating the core instruction (`printf("%d\n", ...)`) with a different value each time. The most general purpose loop in the C programming language is called a `for` loop.

A `for` loop is usually used when the number of iterations (or times around the loop) is known . The basic syntax of a for loop is:

```
for (init; test; inc/dec) {
  statements;
  ...
}
```



Where `init` is a statement that sets up the initial state for the loop, `test` is the condition that has to hold true for the loop to continue (at the start of each cycle) and `inc/dec` is an increment or decrement operation to happen each time the statements inside the loop complete (at the end of each cycle). (Strictly, you can have anything you want in each "slot" in the `for` statement, but the most useful and common uses are those above.)

For example, printing the numbers 1 to 5 could be done with a `for` loop that looks like:

```
#include <stdio.h>

int main(void) {

  for (int i=1; i<=5; i++) {
    printf("%d\n", i);
  }
}
```

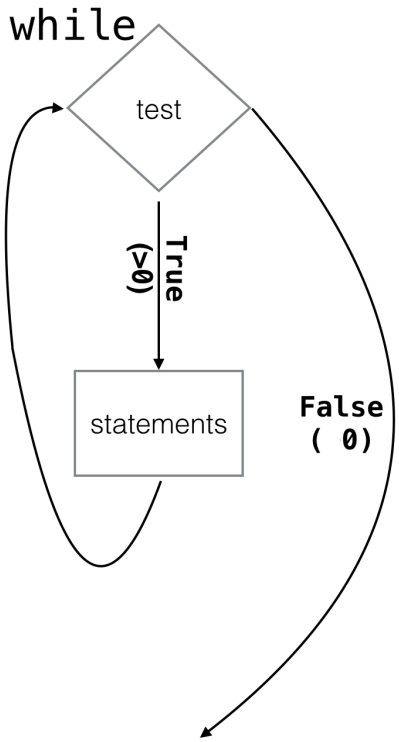
where the init statement starts out the loop by creating a variable called `i`, and setting it equal to 1; the test necessary for the loop to continue is for `i` to be less than or equal to `5`, and `i` is incremented by `1` at the end of each loop.

If we now wanted to print the numbers 1 to 100 we could easily change just the conditional test, whilst leaving the rest unchanged.

In fact, `init`, `test` and `inc/dec`, are each optional, and can be individually left out of a `for` statement definition; one can even write a "minimal" for loop of the form `for( ; ; );`. This type of loop is known as an "infinite loop": as there's no condition to test, the statements inside it will continue to be executed until the program is aborted external or some internal check causes the loop to be exited. In general, care should be taken to make sure conditions in the loop can be met, and it's particularly important to try to avoid modifying the loop variable inside the loop itself (unless you really know what you're doing).

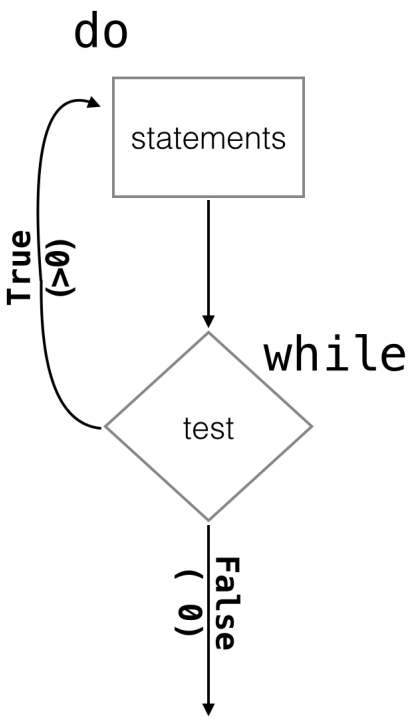
A `while` loop is usually used if a number of iterations is unknown but there is some logical condition that needs to be met in order to exit the loop. The basic syntax of a `while` loop is:

```
while ( condition ) {
  statements;
  ...
}
```



We can also shift the test to the end of the block (so that we always perform the statements exactly once, even if the condition fails) with a `do while` construction:

```
do {
  statements;
  ...
} while (condition);
```



We could write our example of printing the numbers 1 to 5 using a `while` loop:

```
#include <stdio.h>

int main(void) {
  int i;
  i = 1;

  while (i<=5) {
    printf("%d\n", i);
    i++;
  }
}
```

You can see here that the initial condition and increment are separate from the loop construct itself and are part of the logical flow of the program; the design of the `for` is clearly better for this, in terms of keeping all the "loop stuff" together. A better use of a `while` loop might be waiting for enough applications of some complex calculation to achieve a suitably small error:

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

```
int main(void) {
  float error;

  while (error > 0.00001) {
    //Some complex calculation
    error = some_calculation();
  }
}
```

Here, the error is not a "counter" that is external to the functionality of the statements in the loop, but is a direct result of each run through the loop (we assume in this case that the error will at some point become small enough that the loop condition will be met). While we could write a `for` loop to do the same thing, the simplicity of the `while` makes our intent clearer.

Both `for` and `while` loops support the use of the `break` statement, which works similarly to its use in `switch`, immediately jumping outside the loop block to the next statement after it. You can also use the `continue` statement, which instead immediately jumps back up to the top of the loop again (taking any tests necessary to decide if the loop should keep happening on the way). Use these sparingly - the value of a loop construct is that you can get an idea of when it will stop by looking just at the loop statement itself. Adding multiple `break`s, for example, makes it harder to quickly understand what you're doing. On the other hand, especially in `while` loops, it can be useful to have a "backup" `break` in order to deal with exceptional circumstances which don't strictly meet the standards of the main test.

Challenge Questions (2)

Exercise 2: N Green bottles

The popular song English counting song "10 Green Bottles" consists of 10 verses, running:
10 Green Bottles, sitting on the wall
10 Green Bottles, sitting on the wall
and if 1 Green Bottle should accidentally fall...
...there'll be 9 Green Bottles, sitting on the wall.

9 Green Bottles, sitting on the wall...
and so on for subsequent verses (until we reach the special case of 1 Green Bottle, which is the final verse, resulting in "no Green Bottles"). [The concept is similar to other counting-songs in other languages - the number of bottles changes each verse until the end.]

a

Using an appropriate loop, write code to print out the entirety of the 10 Green Bottles song (you may need to use an `if` statement to handle the special case for 1, or place it outside the loop (which is less elegant, but potentially "faster")).

b

By adding appropriate input handling, prompt the user to enter a number of bottles to start from. Check that the number is greater than 0, and is not larger than 99. Alter the loop to count starting from the number the user enters, if it is valid, and print out every verse.

Exercise 3: Squarefree numbers

A "squarefree" number is one which is not divisible by any of the square numbers (4,9,16...).

Write a program which uses the basic method of *trial division* to determine if a provided number is squarefree or not.

Trial division is the process of repeatedly trying to divide the number by all candidate factors (numbers less than it). If it is divisible by at least one, then it cannot be squarefree.

The code should prompt (with text output) for a number, read it in, perform the squarefree test, and then print a suitable response, depending on the result.

After printing the result, the code should ask if you want to perform another test - if so, it should prompt for another number and perform tests. If not, it should exit. (You may use any particular input to test for these options - we suggest 'Y' and 'N'!)

A flowchart for the logic needed for this example is included below; map the design to appropriate structures in C (the grey box surrounds the core logic for the squarefree number test itself).