

A Rudimentary Intro to C programming

Wayne Goddard

School of Computing, Clemson University, 2008

Part 2: Decisions and Repetition

6	For Loops	B1
7	While Loops and Other Loops	B4
8	Ifs	B6
9	More Ifs	B8
10	Booleans	B11
11	Programming Issues	B14

For Loops

Loops are designed for repetition.

6.1 Boolean Conditions and Relational Operators

A *boolean condition* is an expression that is either false or true. A simple boolean condition is a comparison between two values using one of the six *relational operators*:

```
== != < > <= >=
```

The == tests for equality; the != tests for inequality. There must be no space between the characters in these operators.

We will see later how to make more complex conditions.

6.2 The For Loop

A *loop* is used when you have code to be repeated. A *for-loop* is often used when you know how many times something needs to be repeated. This is sometimes called *count*-based looping.

The basic structure is:

```
int index_var;
for ( index_var = init_value; continue_condition; update index_var) {
    body_code ;
}
```

The code that is repeated each time is called the *body*. The variable that keeps track of the number of times through the loop is called the *index variable*.

Execution of the for-loop is as follows.

1. Initialize index variable
2. Check condition. If false then stop.
3. Execute body code.
4. Update index variable.
5. Goto 2.

For example, here is C code to count from 1 up to 10:

```
int i;
for(i=1; i<=10; i++) {
    printf("%d", i);
}
```

Note that braces are optional if the loop body consists of only one statement.

It is important to note that the continue-condition is checked before the body is executed; so it is possible that the body will not be executed at all.

6.3 Overflow

Both the int and the double/float types have finite precision. As a consequence, one can overflow them or get incorrect results.

6.4 Sample Program: factorial.c

This program prints out the factorials up to a given value. Note that it uses the previous value of fact to get the new value of fact.

```
/* calculates first few factorials
 * will overflow
 * wdg 2008
 */
#include <stdio.h>

int main (void) {

    int max, fact, current;

    printf("Input max: ");
    scanf("%d", &max);

    fact = 1;
    for( current = 1; current<= max; current++ ) {
        fact = fact * current;
        printf("%d! is %d\n", current, fact);
    }

    return 0;
}
```

Run the program. If you put in a large value of `max`, the value of `fact` will overflow: an `int` variable can only store values within a fixed range.

6.5 Nested Loops and Sample Program: `squareDraw.c`

A loop within a loop is called a *nested loop*: there is an outer and an inner loop. Here is a program to draw a square of stars, such as

```
*****
*****
*****
*****
*****
```

Notice that the body of the outer `i` loop prints a single row of stars.

```
//This program draws a square of stars - wdg 2008
#include <stdio.h>

//Main function
int main(void) {
    const int SIZE=5;
    const char GLYPH='*';

    int i,j;
    for(i=0; i<SIZE; i++) {
        for(j=0; j<SIZE; j++) {
            printf("%c", GLYPH);
        }
        printf("\n");
    }

    return 0;
}
```

Practice. Adapt the above program to print a triangle of stars, rather than a square.

While Loops and Other Loops

7.1 The While Loop

The second common loop is the *while-loop*. The basic structure is:

```
while ( continuation_condition ) {  
    body_code ;  
}
```

As in the `for` loop, the body might never be executed. As an example, here is code to count up to 10 with a while loop:

```
int i=1;  
while( i<=10 ) {  
    printf("%d", i);  
    i++;  
}
```

Note that initialization occurs before the loop, and update occurs at the end of the body.

The while loop is often used for sentinel-based looping: a *sentinel* is a distinguishing value that tells the loop to stop. For example, the user inputs values until she inputs -1 not to continue.

7.2 Other Loop Stuff

There is also a `do` loop which is occasionally useful. Also occasionally useful are the `break` and `continue` constructs: these allow one to jump out or jump ahead in a loop. We'll do an example later.

7.3 Sample Program: average.c

```
//This program averages some user-supplied numbers
#include <stdio.h>

//Main function
int main(void) {
    int count = 0;
    float value, average;
    float totalSoFar = 0.0;

    printf("Enter first value: (negative to finish) ");
    scanf("%f", &value);
    while( value>=0.0 ) {
        count++;
        totalSoFar += value;
        printf("Enter next value: (negative to finish) ");
        scanf("%f", &value);
    }

    average = totalSoFar/count;
    printf("Average is %.2f\n", average);

    return 0;
}
```

Ifs

8.1 The If Statement

The *if statement* is the primary selection control structure. The format is

```
if ( condition )
    then_code ;
```

The condition is a boolean condition as before. If the condition is true the `then_code` is executed; if the condition is false the `then_code` is skipped. A *statement* is a single piece of code; a *block* is several statements enclosed in braces. Here is an example:

```
if( gradePointAverage >= 3.0) {
    printf("Scholarship retained");
}
```

8.2 Sample Program: listMax.c

Here is a standard way of determining the maximum of a collection of values. The idea is to maintain a variable `maxSoFar` with the maximum value seen so far; this variable is updated each time a new value is seen by comparing it with that value. There is a standard problem of how to initialize the value of `maxSoFar`. This program sidesteps this by assuming that all values are nonnegative, and therefore the maximum is at least zero. Another approach is to initialize the value of `maxSoFar` to the first value that is read.

```
/* calculate maximum of user input
 * Assumes values nonnegative
 * wdg 2008
 */
#include <stdio.h>

int main(void) {

    const int LENGTH = 10;
    printf("Enter %d values\n", LENGTH);
    float maxSoFar = 0.0;
```

```
int i;
float value;
for(i=0; i<LENGTH; i++) {
    printf("Enter value ");
    scanf("%f", &value);
    if( value > maxSoFar )
        maxSoFar = value;
}

printf( "Max is %4.1f\n", maxSoFar );
return 0;
}
```

More Ifs

9.1 The If-Else Statement

A more general version of the *if* statement is the *if-else* structure for executing a statement when the condition is not true. Here the format is

```
if ( condition )
    then_code
else
    otherwise_code
```

Here is an example:

```
if(denom==0)
    printf("Unable to process fraction");
else
    printf("Fraction in decimal is %4.2f", ((float)numer)/denom);
```

Note the “cast” is the previous line: it tells the compiler to treat the `numer` value as a float.

In other words, the `else` part is optional. A common error is to miss the semi-colon before the `else`, or to have a semi-colon after the condition.

9.2 Selection from More Choices

The `otherwise_code` might itself be an *if-else* construct. This is commonly thought of as *if ... else if ... else*. Here is an example:

```
if( waterTemp>100.0 )
    printf("Gas");
else if (waterTemp>0.0 )
    printf("Liquid");
else
    printf("Solid");
```

9.3 The Switch Statement

If you have a long sequence of “*else if*” statements all testing equality of the same variable, then you might prefer a *switch* statement. The following code looks at the

value of `key` and does the appropriate action; more usually a switch is done with an `int` variable. At execution, the value of `key` is examined and control jumps to the appropriate label (given by case statement); `default` is like an `else`.

```
char key = ...
switch ( key ) {
    case 'A':
        printf("alpha"); break;
    case 'B':
        printf("bravo"); break;
    case 'C':
        printf("charlie"); break;
    default:
        printf("zulu"); break;
}
```

The `break` command transfers execution to the first statement after the switch. Without the `break`'s, an `A` would trigger all four `printf`'s.

9.4 Sample Program: `grader.c`

```
// This program converts user-supplied number to letter grade
// wdg 2008
#include <stdio.h>

//Main function
int main(void) {
    float rawGrade;
    char letterGrade;
    printf("Enter grade: ");
    scanf("%f", &rawGrade);
    if( rawGrade<0.0 ) {
        printf("Cannot be negative" );
    }
    else {
        if (rawGrade>=90.0)
            letterGrade = 'A';
        else if (rawGrade>=80.0)
            letterGrade = 'B';
        else if (rawGrade>=70.0)
            letterGrade = 'C';
    }
}
```

```
    else          // rawGrade<70.0
        letterGrade = 'F';
        printf("Value %.2f is worth a %c\n", rawGrade, letterGrade);
    }
    return 0;
}
```

Booleans

10.1 Boolean Variables

A *flag* is a variable that has only two possible values (think true/false or on/off). In C a flag is often simulated by an `int` variable. Note that

C interprets 0 as false and any other value as true.

An example usage could be where we have a loop that can end for many reasons (bad input, problem solved, time elapsed). In this case, we do a loop which repeatedly checks the flag; in the body of the loop there are several places where the flag can be set (to set a flag is to make it true; to clear a flag is to make it false).

Other programming languages such as C++ and Java have a boolean data type built in. This can be achieved in newer versions of C by loading the `stdbool` header.

```
#include <stdbool.h>

bool done = false;
while ( !done ) {
    ...
    if ( ... ) done = true;
    ...
    if ( ... ) done = true;
}
```

10.2 Logical Operators

For more complex boolean conditions, one can combine conditions with one of the *logical operators*. The three most common of these are

```
&& || !
```

The `&&` means *and*: the overall condition is true if both parts are true. The `||` means *or*: the overall condition is true if either or both parts are true (sometimes called the inclusive or). The operator `!` means *not*: it converts false to true and vice versa.

It is useful to know that the logical operators have lower *precedence* than the other operators such as arithmetic and relational. So to test if a given `char` is an upper-case letter, it is sufficient to write as follows (without more brackets):

```
if ( myChar >='A' && myChar <='Z' )
```

It is also sometimes useful to know that the `&&` and `||` are *short-circuit* evaluators: for example, if the first part of an `&&` evaluates to false, so that the result is guaranteed to be false, then the remaining part is not evaluated. For example, to do something with a fraction might write:

```
if ( denom!=0 && num/denom > threshold )
```

If it turns out that `denom` is zero, then the division will not be performed and so one will not trigger a division-by-zero problem.

10.3 Sample Program: `sequence.c`

Here is a program that reads in a sequence from the user and then prints a message as to whether the sequence is strictly increasing, strictly decreasing, or neither.

```
// reads 10 numbers from user and says whether increasing or decreasing
#include <stdio.h>
#include <stdbool.h>

int main( void ) {
    const int LENGTH = 10;
    bool isIncreasing = true, isDecreasing = true;
    float prev, current;
    int i;

    for(i=0; i<LENGTH; i++) {
        printf("Enter value ");
        scanf("%f", &current);
        if( i>0 && current<=prev)
            isIncreasing = false;
        if( i>0 && current>=prev)
            isDecreasing = false;
        prev = current;
    }

    if( isIncreasing )
        printf("Is strictly increasing\n");
    else if( isDecreasing )
        printf("Is strictly decreasing\n");
    else
        printf("Not strictly monotonic\n");
}
```

```
    return 0;  
}
```

Programming Issues

We mention some more issues briefly.

11.1 Style

Good programming means extensive comments and documentation. At the very least, one should always explain the purpose of each variable. You should also strive for a consistent layout and for expressive variable names.

11.2 Testing

Testing should fully exercise your code. You should use both normal data and exceptional data. Pay special attention to outlying data or *boundary* values: 0 as a value, an empty list, etc. Add *watches* or *debug statements* or printouts so that you know what is happening at all times.

11.3 Development

Try to get the program working one part at a time. Print out, print out! Print out partial results, print out what you've just read, print out a message to say where you are, etc. Don't make assumptions.