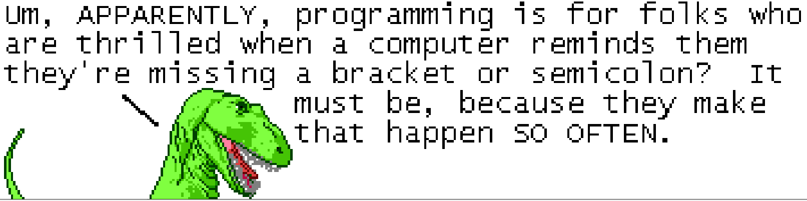
Week 3: Loops, Input and Output
CS211: Programming and Operating Systems

Niall Madden

Wednesday and Thursday, 29+30 Jan, 2020


Um, APPARENTLY, programming is for folks who are thrilled when a computer reminds them they're missing a bracket or semicolon? It must be, because they make that happen SO OFTEN.

Reminder

Lectures

Lecture	Wednesday	15:00–15:50	AC202
Lecture	Thursday	13:00–13:50	AC204
Lab	Friday	09:00–11:50	AdB-G021

Computer labs are a very important part of this course, and attendance is considered mandatory

First lab: this Friday, 1 Feb.

Reminder

1. Selection statements and loops
   - `if` statements
2. for() Loops
   - for-loop arguments
   - Recall... Algorithms
3. `while` - loops
   - `do ... while`
   - Exiting a loop
4. Why not to use `goto`
5. Output: print()
   - plain text
   - Escape Characters
   - Conversion characters
   - Other output functions
6. Input: scanf()

Selection statements and loops

To control the **flow** of a program, one uses

- **Selection Statements:** select a particular execution path. The most important is `if/if else/else` statements. See also, `switch` and, especially, `?:`
- **Iteration statements:** `for`, `while` and `do`
- **jump statements:** `break`, `continue` and `goto`

if statements are used to conditionally execute part of your code.

Structure:

```
if( exprn )
  {
    perform statements if exprn evaluates as
                non-zero
  }
  else
  {
    statements if exprn evaluates as 0
  }
```

Selection statements and loops

Also, if blocks can take the form:

Structure:

```
if( A )
  {
    perform statements if expression A evaluates
                non-zero
  }
  else if( B )
  {
    statements if A is false, but B evaluates as true
  }
  else
  {
    statements if both A and B evaluate as false
  }
```

A trivial example

```
#include <stdio.h>
int main(void )
{
  if (10)
  {
    printf("Non-zero is always true\n");
  }
  if (0)
  {       /* dummy line */    }
  else
    printf("But 0 is never true\n");
  return(0);
}
```

Selection statements and loops

Typically, however, the expressions that `if()` depends on are ***logical expressions***, based on **relational operators**, that must be evaluated.

- `a == 10`
- `c == 'n'`
- `x != 10`
- `z < y`
- `y >= z`

Selection statements and loops

Logical operators, AND, and OR, allow more complex if-statements:

```
if( ( (i%3) == 0) && ( (i%5)==0) )
  printf("%d divisible by 15\n", i);

if( ( (i%3) == 0) || ( (i%5)==0) )
   printf("%d divisible by 3 or by 5\n", i);
```

Selection statements and loops

Examples

01EvenOdd.c ←− link!

```c
// Check Even or Odd
int a=rand()%10; // a is a random number between 0 and 9.
printf("a=%d\n", a);
if ( (a % 2) == 0)
  printf("a is even\n");
else
  printf("a is odd\n");

// Check positive, negative or zero
a=rand()%7-3; // a is a random number between -3 and 3.
printf("a=%d\n", a);
if ( a>0 )
  printf("a is (strictly) positive\n");
else if ( a<0)
  printf("a is (strictly) negative\n");
else
  printf("a is zero\n");
```

for() Loops

```
for( initial val; continuation cond; increment)
```

`for()` is an expression used to execute “loops”: groups of similar tasks to be repeated a certain number of times. It takes three arguments,

- an initial value for the increment variable.
- a condition for continuing the loop.
- instructions on how to modify the increment variable at each iteration.

The tasks to be completed within the loop are contained within curly brackets.

If `{ }` are omitted, then the loop consists only of the line immediately after the `for()` command.

for() Loops

Example (Print a line)

Sometimes we just want a simple operation repeated a fixed number of time. This example just prints a “line” across the screen

```
printf("\n");
for (i=1; i<=60; i++)
  printf("-");
printf("\n");
```

for() Loops

More often, in the body of the loop we use the "***increment variable***" (== "***the loop index"***), as in the following example.

Recall that the ***Fibonacci*** sequence is defined as

$$f_0 = 1, f_1 = 1, \text{ and for } k = 2, 3, \ldots, f_k = f_{k-1} + f_{k-2}.$$

for() Loops

02Fibonacci.c

```c
#include <stdio.h>
int main(void)
{
  int i, Fib[10];
  Fib[0]=1;
  printf("Fib[0] = %d\n", Fib[0]);
  Fib[1]=1;
  printf("Fib[1] = %d\n", Fib[1]);

  for (i=2; i<=9; i++)
  {
     Fib[i] = Fib[i-1] + Fib[i-2];
     printf("Fib[%d] = %d\n", i, Fib[i]);
  }
  return(0);
}
```

for() Loops

Example (Print the odd numbers from 1 to 19)

```
for(i=1; i<= 19; i+=2)
  printf("%d ",i);
```

for() Loops

Example (Count down from 10 to 0)

```
for(i=10; i >=0; i--)
  printf("%d ",i);
```

for() Loops

for-loop arguments

The three arguments to `for` are optional, but the second one is the most important and it is bad practice to omit it.

Example (A bad example)

```
int i=2;
for (; i<10;)
{
  i++;
}
```

Recall... Algorithms

Definition

An **Algorithm** is a finite set of precise instructions for performing a computation or for solving a problem.

Here is an algorithm for finding the maximal element in a finite sequence $a_1, a_2, \ldots, a_n$

Linear Search

```
$m \longleftarrow a_1$
FOR $k = 2$ to $n$
   IF $m < a_k$
      THEN $m \longleftarrow a_k$
   END
END
RETURN $m$
```

Example

Write a short C program that creates a list of 8 randomly chosen integers between 0 and 20, and then finds the largest one.

To solve the problem, we need to do several things:

- Create a random number. This is done using the `rand` function, which requires the `stdlib` header file.
- `rand` produces a number between 0 and 2147483647. Use modulus operator to get one between 0 and 20.
- Use a `for` loop to implement the **linear search algorithm**.

for() Loops

03Largest.c

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

int main(void)
{
  int k, m, a[8];

  printf("\nThe list is: ");
  for (k=0; k<8; k++)   {
    a[k] = rand()%21;
    printf("\t%d", a[k]);
  }
  m = a[0];
  for (k=1; k<8; k++)
    if (m < a[k])
      m = a[k];

  printf("\nThe largest element is: %d\n", m);
  return(0);
```

while - loops

The while loop is probably the simplest loop in C, though not quite as useful as the for loop.

while(*expression*) *statement*

Example

```
while(i < n)
  i*=2;
```

Example

```
i = rand()%100;
while(i < n)
{
  printf("i=%d. Guessing again...\n", i);
  i = rand()%100;
}
```

`while` - loops

These two are equivalent:

```
for (i=0; i<=10; i++)
  sum+=f[i];
```

```
i=0;
while ( i<=10 )
{
  sum+=f[i];
  i++;
}
```

while - loops

This is a trivial loop — it’s statements are never executed:

```
while (0)
{
  // this stuff is ignored
}
```

Whereas the following as an infinite loop:

```
while(1)
{
  printf("We are going to be here a while...");
}
```

while - loops

A do loop is like a while loop, but with the condition for continuation/iteration coming at the end of the block:

```
do
{
    statements
}
while( expression );
```

This is used when we want the statements in the loop to be executed at least once.

04DoWhile.c

```c
#include <stdio.h>

int main(void)
{
  int a;

  do
  {
    printf("Enter an even number : ");
    scanf("%d", &a);
  } while ( a%2 != 0);

  printf("Number %d accepted.\n", a);

  return(0);
}
```

There are (rare) occasions where we might want to

- jump out of a `while`, `for` or `do` loop. This is achieved using `break`.
- skip to the next iteration of the loop, using `continue`.
- jump to another part of a program entirely, using `goto`.

goto

There is ***never*** a good reason to use `goto`. ***Never*** (well, hardly ever)

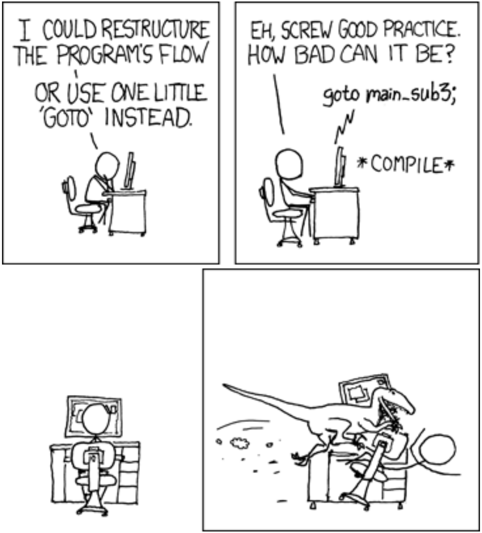
while - loops

Exiting a loop

05BreakContinue.c

```c
#include <stdio.h>

int main(void)
{
  int a;

  for (a=0; a<=100; a++)
  {
    if (a%2 != 0)
      continue;
    printf("a=%d\n", a);

    if (a>=10)
      break;
  };

  return(0);
}
```

Why not to use goto


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

Output: print()

Part of the `standard input/output` library, the `printf()` function is the most commonly used mechanism for sending ***formatted*** output to the screen.

It is unusual because it many actually take an arbitrary number of arguments:

- a format string,
- followed by zero or more variables,

The format string may include

- plain text, to be sent to `stdout`
- ***escape*** characters,
- conversion characters, to tell the system how variables whose values will be displayed. These are actually a bit complicated, and so we won't be able to describe them in full detail.

Output: print()

plain text

To print a simple message, pass you text as the first argument , encapsulated in double quotes:

```
printf("This is not a very interesting example");
```

However, usually this first string argument includes ***escape characters*** and ***conversion characters***

The format string in C may contain a number of "*escape characters*". These are represented with a *backslash*, followed by a single letter, and allow `printf` to "display" commonly used characters, but that don't have easy keyboard representations.

The most important ones are:

- \a Produces a beep or flash (useful when debugging)
- \b Moves the cursor to the last column of the previous line. (Not that useful).
- \f Moves the cursor to start of next page. (not very useful)
- \n New line. The ***most used***
- \r Carriage Return
- \t Horizontal Tab (quite useful when displaying tables of data).
- \v Vertical Tab (not very useful)
- \\\\ Prints single \\
- \\" quotation
- %% Prints %.

Conversion characters

A ***Conversion character*** is a letter that follows a % (percent symbol) and tells `printf` to display the value stored in the variable that is next in its argument list. The most common ones are

- `%c` Single **char**acter (i.e., variable of type `char`,
- `%d` **d**ecimal integer (`int`)
- `%e` floating-point value in `E` ("scientific") notation
- `%f` floating-point value (`float`)
- `%g` Same as `%e` or `%f` format, whichever is shorter
- `%o` octal (base 8) integer
- `%s` String of text (`char` array)
- `%u` Unsigned `int`
- `%x` hexadecimal (base 16) integer

These can also take flags that modify their behaviour.

Output: print()

flags

1. Width specifiers
2. Precision specifiers
3. Input-size modifiers

Examples:

Other output functions

Although `printf` is the most versatile function, there are others for displaying output:

- `putchar`
- `putc`
- `puts`

Input: scanf()

The `scanf()` function is analogous to `printf()`: it will

- read input from standard input,
- format it, as directed by a ***conversion character*** and
- store it in a specified address.

```
int i;
char s;
printf("Enter an integer and a char: ");
scanf("%d %c", &i, &s);

printf("The int is %d, char is %c\n",i,s);
```

Input: scanf()

Example

Write a short C programme that reads a single integer from the keyboard, and checks that it's an even number between 1 and 49 (inclusive).

```
int i;
printf("Enter a positive, even integer less than 50: ");
scanf("%d", &i);

printf("You entered %d", i);
if ((i<=0) || (i>=50) )
  printf(", which is *not* between 1 and 49.\n");
else if ( (i%2) != 0)
  printf(", which is in [1, 49], but is *not* even.\n");
else
  printf(". Thank you.\n ");
```

Input: scanf() Note:

Some other things about scanf:

- We usually call the scanf function is if its return value is void, but it actually returns an integer equal to the number of successful conversions made.
- It has friends fscanf that we'll use for reading from files (in fact scanf is really just fscanf in disguise but with the keyboard as the input "file"), and sscanf used for extracting from strings.
- There are other very useful functions for reading from the standard input stream: getchar, gets

Input Checking

In the last example, we checked that the user inputted that data that was asked for. If we don't include such checks...

NoInputCheck.c

```c
int n, i, list[30];
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
for (i=0; i<n; i++)
  list[i] = rand()%40;
```

While this is OK, it can lead to strange results if the user enters a number less than 1 or greater than 30.

So we should check that the user inputs the data correctly...

Input Checking

We could use an `if` statement to improve this:

IfInputCheck.c

```c
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
if ( (n<1) || (n>30) )
{
  printf("\aError:  number not between 1 and 30\n");
  return(1);
}
```

although it would be better if the user had a chance to enter the data correctly...

Input Checking

So we could ask the user the try entering the data again:

IfInputCheckAgain.c

```
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
if ( (n<1) || (n>30) )
{
  printf("\aError:  number not between 1 and 30\n");
  printf("Enter a number between 1 and 30: ");
  scanf("%d", &n);
}
```

but this only allows the user to make one mistake. Where we have a persistently dumb user, we need to let them try again, and again, and again...

Input Checking

That is easily achieved by using a `while` loop instead of the `if` expression:

WhileInputCheck.c

```c
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
while ( (n<1) || (n>30) )
{
  printf("\aError:  number not between 1 and 30\n");
  printf("Enter a number between 1 and 30: ");
  scanf("%d", &n);
}
```

Now the programme will keep asking the user to enter the number `until` they get it right.

Input Checking

And as described in out previous lecture, we could also use a `do... while` loop. This lets the loop run once, ***before*** checking that the input was correct. If its not, it repeats the loop.

DoWhileInputCheck.c

```
do
{
  printf("Enter a number between 1 and 30: ");
  scanf("%d", &n);
} while ( (n<1) || (n>30) );
```

Exercises

Exercise (Exer 3.1)

Write a short C programme that prompts the user to input an integer, and then uses `scanf` *to read that integer.*

The program should output the value that the user entered and that `scanf` *returns.*

Run the program to check what `scanf` *will return when*

- (i) *the user enters an integer;*
- (ii) *the user enters a float (with decimal part);*
- (iii) *the user enters non-digit character.*

Exercises

Exercise (Exer 3.2)

Write a short C programme that prompts the user to input an integer, i, such that $10 \leqslant i \leqslant 30$.
Use a `while` (or `do... while`) loop so they are repeatedly prompted for this integer until they enter one that is in this range.
Then the program should output an alternating string of zeros and ones of length i.