Loops, Input and Output

CS211: Programming and Operating Systems

Niall Madden

Wednesday and Thursday, 24+25 Feb, 2021

New class times

	Mon	Tue	Wed	Thu	Fri
9 – 10					
10 – 11					
11 – 12					
12 – 1					
1 – 2				✓	
2 – 3					
3 – 4	LAB?		✓		
4 – 5	LAB?				

1. The recorded classes on Wednesdays and Thursdays are unchanged (sorry!).
2. **New lab times: Monday 15:00-17:00**. You aim to attend for an hour. Drop in an out as needed.
3. Little, if any, of the "lab" times will be recorded.
4. All this may all change again towards the end of the semester.
5. **Might switch to Zoom for some classes. Any objections?**

16.50

New class times

1. Part 1: Keywords and Operators
   - Keywords
   - Operators
   - Logic Operators
2. Part 2: Selection statements
   - `if` statements
3. Part 3: Loops
   - `for()`
   - for-loop arguments
   - `while`
   - Exiting a loop
   - Why not to use `goto`
4. Part 4: Output with `printf()`
   - plain text
   - Escape Characters
   - Conversion characters
   - Other output functions
5. Part 5: Input with `scanf()`
   - Input Checking

Wednesday 26th Feb.

Thurs, 25th Feb.

CS211 – Week 3
Loops, Input and Output

Start of ...

PART 1: Keywords and Operators

Part 1: Keywords and Operators

Keywords

C has a set of reserved **keywords**; they cannot be used as variable or function names:

```
auto       double   int       struct
break      else     long      switch
case       enum     register  typedef
char       extern   return    union
const      float    short     unsigned
continue   for      signed    void
default    goto     sizeof    volatile
do         if       static    while
```

Some “new” ones, which are supported by some (but not all) “old” compilers, include

```
restrict   _Bool   _Complex   _Imaginary
```

Operators come in **four** flavours: ***Arithmetic***, ***assignment relational*** and ***logical***.

Arithmetic Operators available in C include:

C Symbol	Definition	Example
+	addition	`c = a + b;`
-	subtraction	`c = a - b;`
*	multiplication	`c = a * b;`
/	division	`c = a / b;`
%	remainder	`c = a % b;`

Unlike Python, there isn't a built-in function for powers or truncating division.

"modulo"

Eg 12 % 5 = 2
18 % 3 = 0

1234 % 100 = 34.
1234 % 10 = 4.
1234 % 1 = 0

Part 1: Keywords and Operators

The Assignment and Arithmetic-Assignment Operators are:

Symbol	Definition	Example
`=`	assignment	`a=b;`
`++`	increment	`a++;`
`--`	decrement	`a--;`
`+=`	addition assignment	`a+=2;`
`-=`	subtraction assignment	`a-=2;`
`*=`	multiplication assignment	`a*=2;`
`/=`	division assignment	`a/=2;`
`%=`	modulo assignment	`a%=2;`

The following is legal, but not encouraged: `i=j=k=0` and is the same as `i = (j = (k = 0))`.

Dont confuse with ==

Eg b=10

a=10

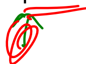
set a to 11

set a=a-1

set a=a+2.

similarly (i=j+=(k--)) is legal, but confusing.

The operator ++ can be used in both ***prefix*** and post-fix form: in prefix form, the increment takes place before the value is used.



01Operators.c

```
int main(void)
{
  int i=1;
  printf("i++ = %d; ", i++);
  printf("++i = %d\n", ++i);
  i=1;
  printf("++i = %d; ", ++i);
  printf("i++ = %d\n", i++);
  return(0);
}
```

postfix

prefix.

A **Relational Operator** tests if some relation holds between two quantities or variables, and evaluates as **true** or **false**.

C Symbol	Maths Symb	Definition
<	$<$	less than
<=	$\leq$	less than or equal.
>		
>=		
==	$=$	test equality ("is equals")
!=	$\neq$	not equals.

These all evaluate as 0 for ***false*** or 1 for true.

Eg $x = (2 > 3);$ sets $x = 0$.

02Logic.c

```
// 02Logic.c; For CS211, Feb 2021. NM
#include <stdio.h>

int main(void)
{
  int i=1, j=2;
  printf("i=%d and j=%d\n", i, j);
  printf("i>j \t\t evaluates as %d\n", i>j);
  printf("++i >= j \t evaluates as %d\n", ++i>=j);

  return(0);
}
```

Try yourself.

Relational operators can be combined into more complex operators, as follows.

C Symbol	Maths Symb	Definition
!	$\sim$ $(\neg)$	not
&&	$\wedge$	AND
||	$\vee$	OR

(inclusive — true if either or both are true).

Eg if (!(a <= b))

same as if (a > b)

Exercise (2.1)

Suppose $x = 2$, $y = 3$ and $z = -5$. Write a C programme that check if the following statements are **true** *or* **false**.

1. $(x > y) \vee (x < y)$.
2. $(x = (y - 1)) \wedge ((y \leqslant x) \vee (y \leqslant z))$.
3. $\neg(y \geqslant x - z) \vee (y \geqslant x + 1)$.

x=2 and y=3
So x<y
is true

false or true

```
if ( (x > y) || (x < y) )
[note - should be true (ie 1)]
printf("Statement 1 is true\n");
```

For division of integers: try

x = (float) (a)/b ;

CS319 – Week 3
Loops, Input and Output

END OF PART 1

CS211 – Week 3
Loops, Input and Output

Start of ...

PART 2: Selection statements

ie if

Part 2: Selection statements

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` ← mostly ignore

`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
  }
```

logical Expression (Something that is true or false).

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
  }
```

Can have zero or more "else if" between if and else.

can be repeated.

optional.

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);
}
```

not zero, so "true".

Part 2: Selection statements

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`

or any combination of these, with && , ||

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);
```

Note:
- "if" body has just one line, so { — } are optional.
- no semi-colon at end of if-line

Example code has 17 lines of comments, etc.

03EvenOdd.c ←– link!

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

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

Rand() returns a large (pseudo)-random number.

CS319 – Week 3
Loops, Input and Output

END OF PART 2

CS211 – Week 3
Loops, Input and Output

Start of ...

PART 3: Loops

(Part 3a)

Now : Part 3b

Part 3: 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.

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");
```

add 1 to i at every step.

init i to 1

Continue while $i \leq 60$

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

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

Eg

$f_2 = 2$

$f_3 = 3$

$f_4 = 5$

$f_5 = 8$

$f_6 = 13$

04Fibonacci.c

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

20   for (i=2; i<=9; i++)
     {
22      Fib[i] = Fib[i-1] + Fib[i-2];
        printf("Fib[%d] = %d\n", i, Fib[i]);
24   }
     return(0);
26 }
```

i=i+1

Example (Print the odd numbers from 1 to 19)

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

note that i increases by 2 at every step.

Example (Count down from 10 to 0)

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

setting $i = i-1$ at every step.

[Stop here]

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 for syntax

for (initialization ; continuation ; increment)

"termination". "step"

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 ⟵ a_1
FOR k = 2 to n
   IF m < a_k
      THEN m ⟵ a_k
   END
END
RETURN m
```

Pseudocode.

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. (ie, mod 21).
- Use a `for` loop to implement the **linear search algorithm**.

#include <stdlib.h>

05Largest.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);
}
```

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;
}
```

These two are equivalent:

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

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

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

zero = false.

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

Whereas the following as an infinite loop:

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

Exercise (do ... while)

There is also a variant called a `do ... while` *loop. Read up on it. Review the example in* `06DoWhile.c` *and work out what it does.*

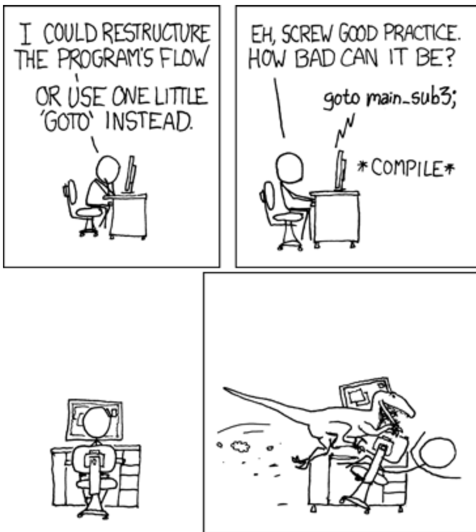
There are (rare) occasions where we might want to

- jump out of a `while`, `for` or `do` loop. This is achieved using `break`.
- skip to the next iteration of the loop, using `continue`. → ie → go back to start.
- See the example in `07BreakContinue.c`
- jump to another part of a program entirely, using `goto`.

`goto`

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

Why not to use `goto`



CS319 – Week 3
Loops, Input and Output

END OF PART 3

CS211 – Week 3
Loops, Input and Output

Start of ...

PART 4: Output with `printf()`

Part 4: Output with `printf()`

aka stdio.

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

[later fprintf -> for files]

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.

used for formatting the line.

format values of variables.

Part 4: Output with `printf()` 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***

Eg printf(" Hello World \n");

↑ new line.

Or printf("\n\n Hello \n world \n");

Give → 2 blank lines

- Hello
- new line
- world — followed by \n.

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 %.

- slightly useful,

Part 4: Output with `printf()` 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 ~ eg 1.0e-2
- `%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.

flags

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

Examples:

```
float x = 3.14159;
```

Code	Output
`printf("x=%f");`	x= 3.14159
`printf("x=%.2f");`	x= 3.14
`printf("x=%8f");`	x= ␣␣3.14159

blank

8 chars in total

Part 4: Output with `printf()` Other output functions

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

- `putchar` → outputs a single char
- `putc` → " " " " (including to file);
- `puts`

→ in a few weeks time.

CS319 – Week 3
Loops, Input and Output

END OF PART 4

CS211 – Week 3
Loops, Input and Output

Start of ...

PART 5: Input with `scanf()`

Part 5: Input with `scanf()`

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

- read input from standard input, (usually keyboard).
- 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);
```

Note the arguments to scanf start with &. This is because scanf changes the value of i & s.

Part 5: Input with `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 ");
```

Some other things about `scanf`:

- We usually call the `scanf` function as 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`

Try r = scanf("%d", &i);

(sets r=1)

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

```
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...

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

IfInputCheck.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...

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...

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

WhileInputCheck.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.

Exercise (do ... while again)

This is a situation where a do ... while would be useful.

1. *Why?*
2. *Write a version using do... while.*

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.