Essentials of programming

This week: Control flow

- Conditionals
- Loops
- Recursion

Example: absolute values

Absolute value of x is

- x if x greater or equal than zero
- $-x$ if x less than zero

Using syntax of mathematics:

$$|x| = \begin{cases} x & \text{, if } x \geq 0 \\ -x & \text{, if } x < 0 \end{cases}$$

Example: absolute values

Absolute value of x is

- x if x greater or equal than zero
- $-x$ if x less than zero

Using syntax of mathematics:

$$|x| = \begin{cases} x & \text{, if } x \geq 0 \\ -x & \text{, if } x < 0 \end{cases}$$

The ifis known as Conditional execution

Conditional Execution is an important programming concept:

- Allows you to write a program that reacts (instead of the same thing every time)

Conditional execution

Previous programs followed one and the same path of execution

- Conditional: alternative path of execution
  - If x greater or equal to zero then ...
  - Else, if x less than equal to zero then ...

- The bit ’x greater or equal to zero’ is known as a *Boolean* expression.
- The values of a Boolean expression is Either
  - True, (for example, $3 > 0$), or
  - False, (for example, $-3 > 0$, not true, therefore false...)
- *True* and *False* are the values of the type *Boolean*

Absolute value in C

```
double absolute( double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}
```

The `x >= 0` is the condition

Absolute value in C

```
double absolute( double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}
```

The `x >= 0` is the condition

The **`if`**, **`else`** and **`{ }`** denote the conditional execution

- If the conditional is true: return x
- If the conditional is not true (false): return -x

Executing absolute value

```
  double absolute(  double x ) {
    if( x >= 0 ) {
      return x ;
    } else {
      return -x ;
    }
  }
⇒ int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

Executing absolute value

```
double absolute(  double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}
int main( void ) {
⇒ printf( "absolute(4) = %f\n", absolute(4) ) ;
  printf( "absolute(-8) = %f\n", absolute(-8) ) ;
  printf( "absolute(0) = %f\n", absolute(0) ) ;
  return 0 ;
}
```

Executing absolute value

```
⇒  double absolute(  double x ) {
     if( x >= 0 ) {   4
       return x ;
     } else {
       return -x ;
     }
   }
   int main( void ) {
     printf( "absolute(4) = %f\n", absolute(4) ) ;
     printf( "absolute(-8) = %f\n", absolute(-8) ) ;
     printf( "absolute(0) = %f\n", absolute(0) ) ;
     return 0 ;
   }
```

Executing absolute value

```
  double absolute(  double x ) {
⇒   if( x >= 0 ) {   4
      return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

Executing absolute value

```
  double absolute(  double x ) {
    if( x >= 0 ) {   4
⇒     return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

Executing absolute value

```
double absolute(  double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}
int main( void ) {
  printf( "absolute(4) = %f\n", absolute(4) ) ;
⇒ printf( "absolute(-8) = %f\n", absolute(-8) ) ;
  printf( "absolute(0) = %f\n", absolute(0) ) ;
  return 0 ;
}
```

```
absolute(4) = 4.000000
```

Executing absolute value

```
⇒ double absolute(  double x ) {
    if( x >= 0 ) {   -8
      return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

```
absolute(4) = 4.000000
```

Executing absolute value

```
  double absolute(  double x ) {
⇒   if( x >= 0 ) {   -8
      return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

```
absolute(4) = 4.000000
```

Executing absolute value

```
double absolute(  double x ) {
  if( x >= 0 ) {   -8
    return x ;
  } else {
⇒   return -x ;
  }
}
int main( void ) {
  printf( "absolute(4) = %f\n", absolute(4) ) ;
  printf( "absolute(-8) = %f\n", absolute(-8) ) ;
  printf( "absolute(0) = %f\n", absolute(0) ) ;
  return 0 ;
}
```

```
absolute(4) = 4.000000
```

Executing absolute value

```
double absolute(  double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}
int main( void ) {
  printf( "absolute(4) = %f\n", absolute(4) ) ;
  printf( "absolute(-8) = %f\n", absolute(-8) ) ;
⇒ printf( "absolute(0) = %f\n", absolute(0) ) ;
  return 0 ;
}
```

```
absolute(4) = 4.000000
absolute(-8) = 8.000000
```

Executing absolute value

```
⇒ double absolute(  double x ) {
    if( x >= 0 ) {   0
      return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

```
absolute(4) = 4.000000
absolute(-8) = 8.000000
```

Executing absolute value

```
  double absolute(  double x ) {
⇒   if( x >= 0 ) {   0
      return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

```
absolute(4) = 4.000000
absolute(-8) = 8.000000
```

Executing absolute value

```
  double absolute(  double x ) {
    if( x >= 0 ) {   0
⇒     return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
    return 0 ;
  }
```

```
absolute(4) = 4.000000
absolute(-8) = 8.000000
```

Executing absolute value

```
  double absolute(  double x ) {
    if( x >= 0 ) {
      return x ;
    } else {
      return -x ;
    }
  }
  int main( void ) {
    printf( "absolute(4) = %f\n", absolute(4) ) ;
    printf( "absolute(-8) = %f\n", absolute(-8) ) ;
    printf( "absolute(0) = %f\n", absolute(0) ) ;
⇒   return 0 ;
  }
```

```
absolute(4) = 4.000000
absolute(-8) = 8.000000
absolute(0) = 0.000000
```

Boolean arithmetic

Relational operators result in a Boolean value

- **<**, **>**, **<=** ($\leq$), **>=** ($\geq$), **==** (equality), **!=** ($\neq$)
- They compare numbers

You can use logical operators to combine booleans:

- **&&** ($\wedge$, AND),
- **||** ($\vee$, OR)
- **!** ($\neg$, NOT)

(also know as *logical* operators)

Boolean operators

- x AND y is true if both x is true and y is true

```
true  AND true  = true
false AND ____  = false
____  AND false = false
```

- x OR y is true if x is true or y is true

```
false OR false = false
true  OR ____  = true
____  OR true  = true
```

- NOT x is true if x is false

```
NOT true  = false
NOT false = true
```

Booleans in C

- C does not have a Boolean type, the two boolean values are represented by numbers
  - 'false' is represented by the number 0
  - 'true' is represented by the number 1
  - conversely, any non-null number is interpreted to mean 'true'

- **`0 && 0`**, **`0 && 1`**, **`1 && 0`**, **`0 || 0`**, **`!1`** are all 0
- **`1 || 1`**, **`1 || 0`**, **`0 || 1`**, **`1 && 1`**, **`!0`** are all 1
- **`!15`** is 0, **`15 && 13`** is 1...

The operators have 'short-cut' semantics:

```
if ( x == 0 || 10 / x > 3 ) ...
```

- Works, even if x equals zero. (Some other languages would bail out)

Using Conditionals

- Example, the power function:
  - $3^5 = 3*(3*(3*(3*(3*1))))$
- In general power is defined as:

$$x^n = \begin{cases} 1 & \text{, if } n = 0 \\ xx^{n-1} & \text{, if } n > 0 \end{cases}$$

Using Conditionals

- Example, the power function:
  - $3^5 = 3 * (3 * (3 * (3 * (3 * 1))))$
- In general power is defined as:

$$x^n = \begin{cases} 1 & \text{, if } n = 0 \\ x x^{n-1} & \text{, if } n > 0 \end{cases}$$

- Using the function in the definition itself is known as a recursive definition.

Power in C

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return x * power( x, n-1 ) ;
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) ) ;
  return 0 ;
}
```

- Similar to the maths $x^n = \begin{cases} 1 & \text{, if } n = 0 \\ xx^{n-1} & \text{, if } n > 0 \end{cases}$

Power in C

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return x * power( x, n-1 ) ;
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) ) ;
  return 0 ;
}
```

- **`n == 0`** is the *termination condition*
- **`power( x, n-1 )`** is the *recursive call*

Power in C, execution

Execution of functions uses a *Stack* to remember the values of parameters

- Arguments of a function are pushed on the stack
- Upon return of a function, arguments are popped off the stack again.
- For every iteration of power, we need a few spaces on the stack

Lets show this in detail

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return x * power( x,n-1 );
  }
}

int main( void ) {
⇒ printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

Executing Power

```
double power( double x, double n ) {
⇒ if( n == 0 ) {
     return 1 ;
   } else {
     return x * power( x,n-1 );
   }
}

int main( void ) {
   printf( "%f\n", power( 3, 2 ) );
   return 0 ;
}
```

x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
⇒   return x * power( x,n-1 );
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
⇒ if( n == 0 ) {
      return 1 ;
    } else {
      return x * power( x,n-1 );
    }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

x:	3
n:	1
x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
⇒   return x * power( x,n-1 );
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

x:	3
n:	1
x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
⇒ if( n == 0 ) {
      return 1 ;
    } else {
      return x * power( x,n-1 );
    }
}

int main( void ) {
    printf( "%f\n", power( 3, 2 ) );
    return 0 ;
}
```

x:	3
n:	0
x:	3
n:	1
x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
⇒   return 1 ;
  } else {
    return x * power( x,n-1 );
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

x:	3
n:	0
x:	3
n:	1
x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return x * power( x,n-1 );
  }         ⇑
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

1

x:	3
n:	1
x:	3
n:	2

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return x * power( x,n-1 );
  }         ⇑
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

3

x: 3

n: 2

Executing Power

```
double power( double x, double n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return x * power( x,n-1 );
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 2 ) );
  return 0 ;
}
```

9

Concluding Conditionals

- The type *Boolean* has two values: *true* and *false* (1 and 0 in C)
- Relational expressions **`a<b`**, **`a>=b`**, ... result in a *Boolean*
- Boolean expressions (**`a && b`**, **`(a != 0 && b>0) || ...`**) result in a *Boolean*

Conditional execution:

- Choose an execution path depending on a boolean
- **`if( ) { ... } else { ... }`** in C

Recursion:

- Just express a function using itself.

Next lecture: Loops

Iteration

What if some operation has to be repeated? For example:

- Power: $y^n = y \times y \times y \times y \times .... \times y$
- Factorial $y! = 1 \times 2 \times 3 \times 4 \times ... \times y - 1 \times y$

We have seen recursion as a solution.

- Some programming languages provide other constructs.
- Imperative languages provide *Loops*:
  - for-loop,
  - while-loop, until-loop
  - repeat-loop, do-loop,
  - forever-loop

Before diving into loops we will discuss variables.

Reusing Cells, Variables

Until now, only *Values* were used

- Values are calculated once,
- Once calculated, values do not change

C has *Variables*

- A Variable can *contain* a value.
- A Variable can be *overwritten* to contain another value
- A Variable *is* a memory cell with a *name*
  - Cell can be in many places (stack, heap, global, ...)

Is this a nice mechanism?

- Very powerful.
- Sometimes nasty, examples are shown later on

Changing a Variable

Variables are changed using an “assignment”, for example

```
int i, j ;
i = 4 ;     /* This assigns 4 to i */
i = i + 1 ; /* And this assigns 5 to i */
j = i + 1 ; /* And this assigns 6 to j */
```

Note that the order of statements is important.

Suppose that we change the order:

```
i = 4 ;     /* This assigns 4 to i */
j = i + 1 ; /* And this assigns 5 to j */
i = i + 1 ; /* And this assigns 5 to i */
```

Assignment, example function

Function to calculate x^8

```
int powereight( int x ) {
  int result ;
  result = x * x ;  /* xˆ2 */
  result = result * result ; /* xˆ4 */
  result = result * result ; /* xˆ8 */
  return result ;
}
```

Calculation of **`powereight(3)`** ?

Assignment, example function

Function to calculate x^8

```
  int powereight( int x ) {
⇒   int result ;
    result = x * x ;  /* x^2 */
    result = result * result ; /* x^4 */
    result = result * result ; /* x^8 */
    return result ;
  }
```

Calculation of **powereight(3)** ?

x : $\boxed{\quad\quad 3}$

Assignment, example function

Function to calculate x^8

```
   int powereight( int x ) {
     int result ;
⇒    result = x * x ;  /* x^2 */
     result = result * result ; /* x^4 */
     result = result * result ; /* x^8 */
     return result ;
   }
```

Calculation of **powereight(3)** ?

x :	**3**
result :	**.**

Assignment, example function

Function to calculate x^8

```
   int powereight( int x ) {
     int result ;
     result = x * x ;  /* xˆ2 */
⇒    result = result * result ; /* xˆ4 */
     result = result * result ; /* xˆ8 */
     return result ;
   }
```

Calculation of **powereight(3)** ?

x :	3
result :	9

Assignment, example function

Function to calculate x^8

```
int powereight( int x ) {
  int result ;
  result = x * x ;  /* x^2 */
  result = result * result ; /* x^4 */
⇒ result = result * result ; /* x^8 */
  return result ;
}
```

Calculation of **powereight(3)** ?

x :	3
result :	81

Assignment, example function

Function to calculate x^8

```
int powereight( int x ) {
  int result ;
  result = x * x ;  /* x^2 */
  result = result * result ; /* x^4 */
  result = result * result ; /* x^8 */
⇒  return result ;
}
```

Calculation of **`powereight(3)`** ?

`x` :	`3`
`result` :	`6561`

Assignment, example function

Function to calculate x^8

```
int powereight( int x ) {
  int result ;
  result = x * x ;  /* xˆ2 */
  result = result * result ; /* xˆ4 */
  result = result * result ; /* xˆ8 */
  return result ;
}
```

Calculation of **`powereight(3)`** ?

Answer: 6561.

Loops

- An assignment is especially useful in conjunction with a *Loop*
- First discuss the simplest loop, a *while*, which is used to execute some code until a condition fails (the *continuation condition*)
- A while loop has a *condition* and a *body*

```
while( condition ) {
  body ;
}
```

- The body is executed while the condition is true

Use of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

The syntax is `while(    ) {    }`

Use of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

The syntax is **`while(   ) {    }`**

The continuation condition is **`n > 1`**

Use of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

The syntax is **`while(   ) {    }`**

The continuation condition is **`n > 1`**

The body is **`prod = n * prod; n = n - 1;`**

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
⇒  int prod = 1;
   while( n > 1 ) {
     prod = n * prod ;
     n = n - 1 ;
   }
   return prod ;
}
```

prod:	1
n:	3

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
⇒ while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

prod:	1
n:	3

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
⇒   prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

prod:	1
n:	3

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
   int factorial( int n ) {
     int prod = 1;
     while( n > 1 ) {
       prod = n * prod ;
⇒      n = n - 1 ;
     }
     return prod ;
   }
```

prod:	~~1~~ 3
n:	3

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
  int factorial( int n ) {
    int prod = 1;
⇒   while( n > 1 ) {
      prod = n * prod ;
      n = n - 1 ;
    }
    return prod ;
  }
```

prod: ~~1~~ 3

n: ~~3~~ 2

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
⇒   prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

prod:	~~1~~ 3
n:	~~3~~ 2

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
    prod = n * prod ;
⇒   n = n - 1 ;
  }
  return prod ;
}
```

prod: $\cancel{1}\ \cancel{3}\ 6$

n: $\cancel{3}\ 2$

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
⇒ while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

prod:	~~1~~ ~~3~~ 6
n:	~~3~~ ~~2~~ 1

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
   int prod = 1;
   while( n > 1 ) {
     prod = n * prod ;
     n = n - 1 ;
   }
⇒  return prod ;
}
```

prod: $\cancel{1}\ \cancel{3}\ 6$

n: $\cancel{3}\ \cancel{2}\ 1$

Calculate **`factorial( 3 )`** ?

Execution of the while loop

```
int factorial( int n ) {
  int prod = 1;
  while( n > 1 ) {
    prod = n * prod ;
    n = n - 1 ;
  }
  return prod ;
}
```

Calculate **`factorial( 3 )`** ?

Answer: 6.

Finishing the while loop

The condition of a while loop is a “continuation” condition, not a termination condition as in recursion (continuation = !termination)

- The loop executes while the condition evaluates to true.
- The loop terminates when the condition is false.
- A while loop is used to execute a bit of code until a condition fails
- The while body is not executed if the condition is initially false

C has two other loops:

- A do-loop is similar to a while loop, but evaluates the condition *after* the body of the loop
- A for loop is used to execute a loop a number of times, e.g. 10. In C the for is similar to the while, other languages have real for-loops

The do-loop

Factorial with a do-loop

```
int factorial( int n ) {
  int prod = 1;
  do {
    prod = n * prod ;
    n = n - 1 ;
  } while( n > 1 ) ;
  return prod ;
}
```

Question: What would **`factorial( 0 )`** be?

Answer: Because the body is executed before the condition is evaluated, the answer is wrong.

The do-loop

- The condition of a do-loop is evaluated after the body of the loop:
- Useful if the body must be executed at least once
  - Useful if the condition is undefined before the body of the loop.
  - Not used too often. 98% of the loops is a **`while`** or a **`for`**
- Note:
  - The condition is a “continuation” condition not a termination condition (unlike Modula, Pascal)
  - The term do-loop has a different meaning in Fortran (where a do-loop is actually a real for-loop... sic.)

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i=2 ; i<=n ; i=i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

The for-loop executes a loop while maintaining a counter

- `i` in this case

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i=2 ; i<=n ; i=i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

The for-loop executes a loop while maintaining a counter

- `i` in this case

Syntax:

- **`for(`** initialisation **`;`** condition **`;`** increment**`) {   }`**

Execution pattern:

- initialise ; test ; body ; inc ;test ; body ; inc

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i = 2 ;
              i <= n ;
                      i = i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

Evaluate `factorial(3)`

The for-loop

```
  int factorial( int n ) {
⇒   int i, prod = 1;
    for( i = 2 ;
                 i <= n ;
                         i = i+1 ) {
      prod = i * prod ;
    }
    return prod ;
  }
```

n: 3

Evaluate **`factorial(3)`**

The for-loop

```
   int factorial( int n ) {
     int i, prod = 1;
⇒    for( i = 2 ;
                  i <= n ;
                          i = i+1 ) {
       prod = i * prod ;
     }
     return prod ;
   }
```

prod:	1
i:	.
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
   int factorial( int n ) {
     int i, prod = 1;
     for( i = 2 ;
⇒                 i <= n ;
                          i = i+1 ) {
       prod = i * prod ;
     }
     return prod ;
   }
```

prod:	1
i:	2
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
  int factorial( int n ) {
    int i, prod = 1;
    for( i = 2 ;
                i <= n ;
                        i = i+1 ) {
⇒     prod = i * prod ;
    }
    return prod ;
  }
```

prod:	1
i:	2
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
  int factorial( int n ) {
    int i, prod = 1;
    for( i = 2 ;
                 i <= n ;
                         i = i+1 ) {
      prod = i * prod ;
⇒   }
    return prod ;
  }
```

prod:	~~1~~ 2
i:	2
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i = 2 ;
               i <= n ;
                       i = i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

prod:	~~1~~ 2
i:	2
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
   int factorial( int n ) {
     int i, prod = 1;
     for( i = 2 ;
⇒                i <= n ;
                        i = i+1 ) {
       prod = i * prod ;
     }
     return prod ;
   }
```

prod:	~~1~~ 2
i:	~~2~~ 3
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i = 2 ;
             i <= n ;
                    i = i+1 ) {
⇒    prod = i * prod ;
  }
  return prod ;
}
```

prod:	~~1~~ 2
i:	~~2~~ 3
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i = 2 ;
             i <= n ;
                     i = i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

prod:	$\cancel{1}$ $\cancel{2}$ 6
i:	$\cancel{2}$ 3
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
   int factorial( int n ) {
     int i, prod = 1;
     for( i = 2 ;
                  i <= n ;
⇒                        i = i+1 ) {
       prod = i * prod ;
     }
     return prod ;
   }
```

prod:	~~1~~ ~~2~~ 6
i:	~~2~~ 3
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i = 2 ;
               i <= n ;
                       i = i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

prod:	$\cancel{1}$ $\cancel{2}$ 6
i:	$\cancel{2}$ $\cancel{3}$ 4
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
  int factorial( int n ) {
    int i, prod = 1;
    for( i = 2 ;
                 i <= n ;
                         i = i+1 ) {
      prod = i * prod ;
    }
⇒   return prod ;
  }
```

prod:	$\cancel{1}$ $\cancel{2}$ 6
i:	$\cancel{2}$ $\cancel{3}$ 4
n:	3

Evaluate **`factorial(3)`**

The for-loop

```
int factorial( int n ) {
  int i, prod = 1;
  for( i = 2 ;
             i <= n ;
                    i = i+1 ) {
    prod = i * prod ;
  }
  return prod ;
}
```

Evaluate **`factorial(3)`**

Answer: 6.

For loop versus while loop

Actually:

```
for( A ; C ; I ) {
  B ;
}
```

Is equal to:

```
A ;
while( C ) {
  B ;
  I ;
}
```

The elements of the for-loop

The initialisation, termination and iteration part can be anything:

```
for( x=0 ; City > United ; y=Q*7/i )
```

- Is as legal as any other for-loop...

- Common use:

```
for( count = 0 ; count < n ; count = count + 1 ) {
  ...
}
```

or

```
for( count=n-1 ; count >= 0; count = count - 1 ) {
  ...
}
```

The elements of the for-loop

The initialisation, termination and iteration part can be anything:

```
for( x=0 ; City > United ; y=Q*7/i )
```

- Is as legal as any other for-loop...

- Common use:

```
for( count = 0 ; count < n ; count = count + 1 ) {
   ...
}
```

or

```
for( count=n-1 ; count >= 0; count = count - 1 ) {
   ...
}
```

The elements of the for-loop

The initialisation, termination and iteration part can be anything:

```
for( x=0 ; City > United ; y=Q*7/i )
```

- Is as legal as any other for-loop...

- Common use:

```
for( count = 0 ; count < n ; count++ ) {
   ...
}
```

or

```
for( count=n-1 ; count >= 0; count-- ) {
   ...
}
```

Breaking out of loops

Sometimes loops need to be cut short.

- You can break out of any loop with the **`break ;`** statement.
- A **`break`** will cause the loop to be aborted immediately.
- The next statement executed is the statement following the loop

Compare this with **`return`**

- A **`return`** will cause a function to be aborted immediately
- Control returns to the calling function.

Conditionals, Recursion, Loops

Iteration:

- Recursion `func( .. ) { ... func( ... ) ... }`
- While condition holds: `while( ... ) { x = ... ; ... }`
- Do-While: same but executed at least once.
- Loop with counter:
  `for( i=0 ; i<n ; i++ ) { x = ... ; ... }`
- Break out of any loop with `break ;`

These are almost all flow control mechanisms