CHAPTER

12

C Functions

Function is a group of statements that together perform a task. Every C program has at least one function, which is **main(),** and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The

parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body:** The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

A **function declaration** tells the compiler about a function name and how to call the function. The actual **body of the function** can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are **two ways** that arguments can be passed to a function:

Function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y)
{
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

Now, let us call the function swap() by passing actual values as in the following example:

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100

Before swap, value of b :200
```

```
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

Function call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y)
{
   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put x into y */

   return;
}
```

Let us call the function swap() by passing values by reference as in the following example:

```
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
    * &a indicates pointer to a ie. address of variable a and
    * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
```

```
   return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

CHAPTER

13

C Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called **local** variables,
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which is called **formal** parameters.

Let us explain what are **local** and **global** variables and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using **local variables**. Here all the variables a, b and c are local to **main()** function.

```
#include <stdio.h>

int main ()
{
  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A **global variable** can be accessed by any function. That is, a **global variable** is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <stdio.h>

/* global variable declaration */
int g;

int main ()
{
  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main ()
{
  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of g = 10
```

Formal Parameters

Function parameters, so called **formal parameters**, are treated as local variables within that function and they will take preference over the global variables. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main ()
{
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type	Initial Default Value
int	0

char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly otherwise, your program may produce unexpected results because uninitialized variables will take some garbage value already available at its memory location.

Chapter 3
Programming with Recursion

(Version of 16 November 2005)

3.1. Examples

Factorial (revisited from Section 2.13)

Program (`fact.sml`)

```
fun fact n =
    if n < 0 then error "fact: negative argument"
    else if n = 0 then 1
    else n * fact (n-1)
```

Useless test of the error case at each recursive call!
Hence we introduce an auxiliary function,
and can then use pattern matching in its declaration:

```
local
    fun factAux 0 = 1
      | factAux n = n * factAux (n-1)
in
    fun fact1 n =
        if n < 0 then error "fact1: negative argument"
        else factAux n
end
```

In fact1: pre-condition verification (*defensive programming*)
In factAux: no pre-condition verification
Function factAux is not usable directly: it is a local function

Exponentiation

Specification

function expo x n
TYPE: real $\longrightarrow$ int $\longrightarrow$ real
PRE: n $\geq$ 0
POST: x^n

Construction

Error case: n $<$ 0: produce an error message

Base case: n = 0 : return 1

General case: n $>$ 0 :
return $x^n = x * x^{n-1} =$ x $*$ expo x (n$-$1)

Program (`expo.sml`)

```
local
    fun expoAux x 0 = 1.0
      | expoAux x n = x * expoAux x (n-1)
in
    fun expo x n =
        if n < 0 then error "expo: negative argument"
        else expoAux x n
end
```

Sum

Specification

function sum a b
TYPE: int $\longrightarrow$ int $\longrightarrow$ int
PRE: (none)
POST: $\sum_{a \leq i \leq b} i$

Construction

Base case: a > b : return 0

General case: a $\leq$ b :

return $\sum_{a \leq i \leq b} i$ = a + $\sum_{a+1 \leq i \leq b} i$ = a + sum (a+1) b

Program (`sum.sml`)

```
fun sum a b =
    if a > b then 0
    else a + sum (a+1) b
```

3.2. Induction

Objectives of a construction method

- Construction of programs that are *correct* with respect to their specifications
- A correctness proof must be easily *derivable* from the construction process

Induction is the basic tool for the construction and proof of recursive programs:

- Simple induction
- Complete induction

Simple induction: Example 1

Objective

Prove $S(n)$: $\sum_{0 \le i \le n} 2^i = 2^{n+1} - 1$ for all $n \ge 0$

Base

Prove $S(0)$: $\sum_{0 \le i \le 0} 2^i = 2^{0+1} - 1$

Proof:

$$\begin{aligned} 2^0 &= 2^1 - 1 \\ 1 &= 2 - 1 \end{aligned}$$

Induction

Hypothesis: $S(n)$ is true, for some $n \ge 0$

Prove $S(n+1)$: $\sum_{0 \le i \le n+1} 2^i = 2^{(n+1)+1} - 1$

Proof:

$$\begin{aligned} \sum_{0 \le i \le n+1} 2^i &= \sum_{0 \le i \le n} 2^i + 2^{n+1} \\ &= 2^{n+1} - 1 + 2^{n+1} \\ &= 2^{n+2} - 1 \end{aligned}$$

Conclusion

$S(n)$ is true for all $n \ge 0$

Simple induction: Example 2

Program (`expo.sml`)

```
local
    fun expoAux x 0 = 1.0
      | expoAux x n = x * expoAux x (n-1)
in
    fun expo x n =
        if n < 0 then error "expo: negative argument"
        else expoAux x n
end
```

Correctness proof

Theorem: expo x n $= x^n$ for every real x and integer n ≥ 0

Correctness proof

Theorem: `expo x n` $= x^n$ for every real `x` and integer `n` ≥ 0

Proof by induction on `n`

Error case: `n` < 0
Invalid input, the program stops with an error message

For the other cases,
it suffices to show that `expoAux x n` $= x^n$,
for every real `x` and integer `n` ≥ 0

Base case: `n` $= 0$
We have that `expoAux x n` reduces to `1`, which is x^n

General case (induction): `n` > 0
Hypothesis: `expoAux x (n−1)` $= x^{n-1}$
Now, `expoAux x n` reduces to `x` $*$ `expoAux x (n−1)`,
which reduces (by the induction hypothesis) to `x` $* x^{n-1}$,
which is x^n

The essence of the proof was present during the construction process!

Simple induction: Principles

Objective

Prove $S(n)$ for all integers $n \geq a$

In practice, we often have $a = 0$ or $a = 1$

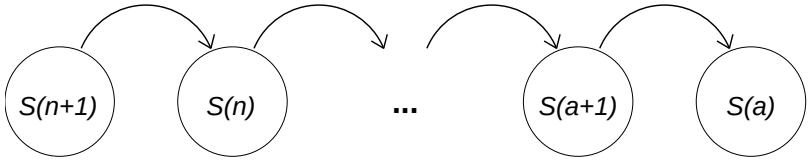
Base

- Prove $S(a)$

Induction

- Make the induction hypothesis, for some $n \geq a$, that $S(n)$ is true
- Prove $S(n + 1)$

That is, prove $S(n) \Rightarrow S(n + 1)$, for some $n \geq a$



Conclusion

$S(n)$ is true for all integers $n \geq a$

Simple induction: Justification

Why is $S(n)$ true for any integer value $n \geq a$?

Proof by iteration

$S(a)$ is true, so $S(a+1)$ is true, ...,
thus $S(n-1)$ is true, hence $S(n)$ is true

Proof by contradiction

Suppose $S(n)$ is false for some n

Let j be the smallest integer such that $S(j)$ is false:

- If $j = a$, then the base is incorrect, as $S(a)$ is false
- If $j > a$, then the induction is incorrect,
  as $S(j)$ is false, hence $S(j-1)$ must be true,
  but $S(j-1) \Rightarrow S(j)$ is false then

Complete induction

For proving the correctness of a program for the function $f\,\mathsf{n}$
simple induction can only be used when
all recursive calls are of the form $f\,(\mathsf{n}-1)$

If a recursive call is of the form $f\,(\mathsf{n}-b)$
where b is an arbitrary positive integer,
then one must use *complete induction*

Complete induction: Principles

Objective

Prove $S(n)$ for all integers $n \geq a$

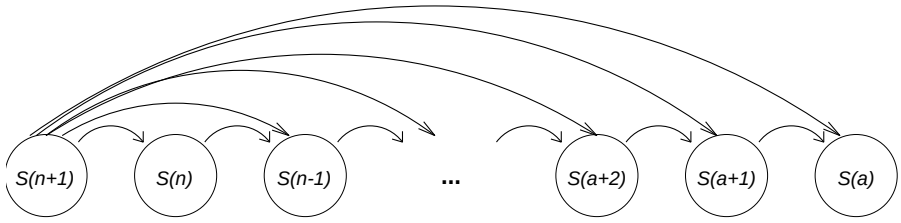
In practice, we often have $a = 0$ or $a = 1$

Base

- Prove $S(a)$

Induction

- Make the induction hypothesis, for some $n \geq a$, that $S(k)$ is true for *every* k such that $a \leq k \leq n$
- Prove $S(n + 1)$



Conclusion

$S(n)$ is true for all integers n such that $n \geq a$

3.3. Construction methodology

Objective

Construction of an SML program computing the function:

$$f(x) : D \rightarrow R$$

given its specification S

Methodology

1. Choice of a variant

A case analysis is done on a numeric variant expression:
let a be the chosen variant, of type A, and
let $A' \subseteq A$ be the lower-bounded set of possible values of a, considering its type A and the pre-condition of S

2. Handling of the error cases

What if $a \notin A'$?
Defensive programming: raise an exception
Otherwise: *assume* the caller established the pre-condition

3. Handling of the base cases

For all the minimal values of a,
directly (without recursion) express the result in terms of x

4. Handling of the general case

When a has a non-minimal value,
investigate how the results of one or more recursive calls
can be combined with the argument
so as to obtain the desired overall result, such that:

1. Recursive calls are on a', of type A, such that $a' < a$
2. Recursive calls satisfy the pre-condition of S

State all this via an expression computing the result

Correctness

If a program is constructed using this methodology,
then it is correct with respect to its specification
(as long as the cases are correctly expressed)

This methodology makes the following hypotheses:

- The general case *can* be expressed using recursion
- The resolution of the problem does *not* involve unspecified and/or unimplemented auxiliary problems
- The set A' has a *lower* bound

A more general program construction methodology

1. Specification
2. Choice of a variant
3. Specification of auxiliary problems (if any)
4. Handling of the error cases
5. Handling of the base cases
6. Handling of the general case
7. Program
8. Construction of programs for the auxiliary problems (if any), following the same methodology again!

Some steps can be useless for the solving of some problems:

- Steps 2 and 5 are useless for non-recursive programs
- Steps 3 and 8 are useless if there are no auxiliary problems

3.4. Forms of recursion

Up to now:

- *One* recursive call
- Some variant is decremented by *one*

That is: simple recursion
(construction process by simple induction)

Forms of recursion

- Simple recursion
- Complete recursion
- Multiple recursion
- Mutual recursion
- Nested recursion
- Recursion on a generalised problem

Complete recursion

Example 1: integer division (quotient and remainder)

Specification

function intDiv a b
TYPE: int $\longrightarrow$ int $\longrightarrow$ (int $*$ int)
PRE: a $\geq$ 0 and b $>$ 0
POST: (q,r) such that a = q $*$ b + r and 0 $\leq$ r $<$ b

Construction

Variant: a

Error case: a $<$ 0 or b $\leq$ 0 : produce an error message

Base case: a $<$ b : since a = 0 $*$ b + a, return (0,a)
This covers more than the minimal value of a (namely 0)!

General case: a $\geq$ b :
since a = q $*$ b + r iff (a$-$b) = (q$-$1) $*$ b + r,
the call intDiv (a$-$b) b will give q$-$1 and r

Program (`intDiv.sml`)

```
fun intDiv a b =
let
    fun intDivAux a b =
        if a < b then (0,a)
        else let val (q1,r1) = intDivAux (a−b) b
             in (q1+1,r1) end
in
    if a < 0 orelse b <= 0 then error "intDiv: invalid argument"
    else intDivAux a b
end
```

Necessity of the induction hypothesis not only for `a−1`, but actually for *all* values less than `a`: complete induction!

Example 2: exponentiation (revisited from Section 3.1)

Specification

function fastExpo x n
TYPE: real $\longrightarrow$ int $\longrightarrow$ real
PRE: n $\geq$ 0
POST: x^n

Construction

Variant: n

Error case: n $<$ 0: produce an error message

Base case: n = 0 : return 1

General case: n $>$ 0 :
if n is even, then return $x^{n \text{ div } 2} * x^{n \text{ div } 2}$
otherwise, return $x * x^{n \text{ div } 2} * x^{n \text{ div } 2}$

Program (`expo.sml`)

```
fun fastExpo x n =
let
    fun fastExpoAux x 0 = 1.0
      | fastExpoAux x n =
          let val r = fastExpoAux x (n div 2)
          in  if even n then r * r
              else x * r * r
          end
in
    if n < 0 then error "fastExpo: negative argument"
    else fastExpoAux x n
end
```

Complete recursion,
but the size of the input is divided by 2 each time!

Complexity

Let $C(n)$ be the number of multiplications made
(in the worst case) by **fastExpo**:

$$C(0) = 0$$
$$C(n) = C(n \text{ div } 2) + 2 \quad \text{for } n > 0$$

One can show that $C(n) = O(\log n)$

Multiple recursion

Example: the Fibonacci numbers

Definition

fib (0) = 1
fib (1) = 1
fib (n) = fib (n−1) + fib (n−2) for n > 1

Specification

function fib n
TYPE: int → int
PRE: n ≥ 0
POST: fib (n)

Program `(fib.sml)`

Variant: n

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib (n−1) + fib (n−2)
```

- Double recursion
- Inefficient: multiple recomputations of the same values!

Mutual recursion

Example: recognising even integers and odd integers

Specification

function even n
TYPE: int $\longrightarrow$ bool
PRE: n $\geq$ 0
POST: **true** if n is even
false otherwise

function odd n
TYPE: int $\longrightarrow$ bool
PRE: n $\geq$ 0
POST: **true** if n is odd
false otherwise

Program `(even.sml)`

Variant: n

```
fun even 0 = true
  | even n = odd (n−1)
and odd 0 = false
  | odd n = even (n−1)
```

- Simultaneous declaration of the functions
- Global correctness reasoning

Nested recursion and lexicographic order

Example 1: the Ackermann function

Definition

For m, n ≥ 0:
acker (0,m) = m+1
acker (n,0) = acker (n−1, 1) for n > 0
acker (n,m) = acker (n−1, acker (n, m−1)) for n, m > 0

Program `(acker.sml)`

Variant: the pair (n,m)

```
fun acker 0 m = m+1
  | acker n 0 = acker (n−1) 1
  | acker n m = acker (n−1) (acker n (m−1))
```

where
$(n', m') <_{lex} (n, m)$ iff $n' < n$ or $(n' = n$ and $m' < m)$
This is called a *lexicographic order*

- The function acker always terminates
- It is not a primitive-recursive function,
  so it is impossible to estimate an upper bound for acker n m

Example 2: Graham's number, the "largest" number

Definition

Operator $\uparrow^n$ (invented by Donald Knuth):

$$a \uparrow^1 b = a^b$$
$$a \uparrow^n b = a \uparrow^{n-1} (b \uparrow^{n-1} b) \quad \text{for } n > 1$$

Program (`graham.sml`)

Variant: `n`

```
fun   opKnuth 1 a b = Math.pow (a,b)
  |   opKnuth n a b = opKnuth (n−1) a (opKnuth (n−1) b b)
```

```
-   opKnuth 2 3.0 3.0 ;
  val it = 7.62559748499E12 : real
-   opKnuth 3 3.0 3.0 ;
  ! Uncaught exception: Overflow
```

Graham's number is `opKnuth 63 3.0 3.0`
It is in the Guiness Book of Records!

Recursion on a generalised problem

Example: recognising prime numbers

Specification

function prime n
TYPE: int $\longrightarrow$ bool
PRE: n $>$ 0
POST: **true** if n is a prime number
false otherwise

Construction

It is impossible to determine whether n is prime
via the reply to the question "is $n-1$ prime"?
It seems impossible to directly construct a recursive program

We thus need to find another function:

- that is more general than **prime**, in the sense
  that **prime** is a particular case of this function
- for which a recursive program can be constructed

Specification of the generalised function

function divisors n low up
TYPE: int $\longrightarrow$ int $\longrightarrow$ int $\longrightarrow$ bool
PRE: n, low, up $\geq$ 1
POST: **true** if n has no divisors in $\{$low, ..., up$\}$
false otherwise

Construction of a program for the generalised function

Variant: up $-$ low $+$ 1, which is the size of $\{$low, ..., up$\}$

Base case: low $>$ up :
return **true** because the set $\{$low, ..., up$\}$ is empty

General case: low $\leq$ up :
if n is divisible by low, then return **false**
otherwise, return whether n has a divisor in $\{$low+1, ..., up$\}$

Construction of a program for the original function

The function prime is a particular case of the function divisors, namely when low is 2 and up is n$-$1
One can also take up as $\lfloor\sqrt{n}\rfloor$, and this is more efficient

Program (`prime.sml`)

```
fun divisors n low up =
    low > up  orelse
    (n mod low) <> 0  andalso  divisors n (low+1) up

fun prime n =
    if n <= 0 then error "prime: non-positive argument"
    else if n = 1 then false
    else divisors n 2 (floor (Math.sqrt (real n)))
```

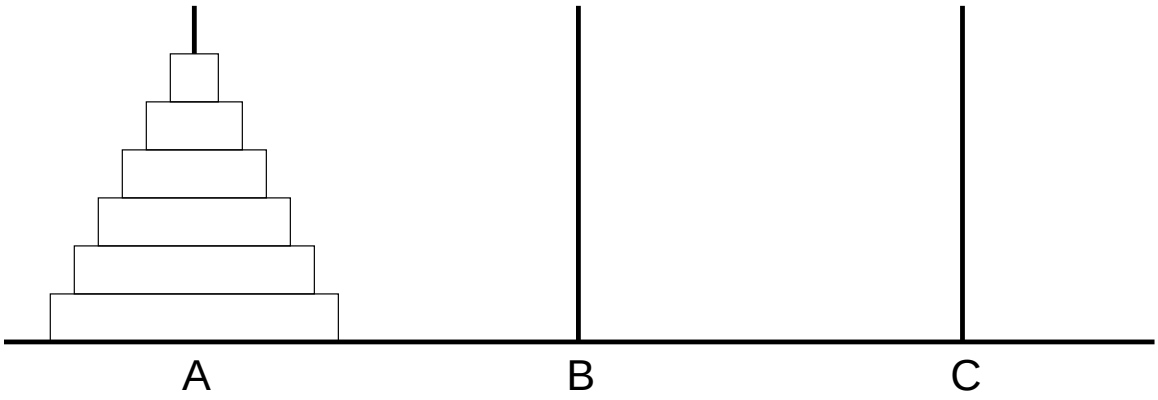
- The function **divisors** has not been declared as local to the function **prime** because it can be useful for other problems
- The discovery of **divisors** requires imagination and creativity
- There are some standard methods of generalising problems:
  - descending generalisation (aka accumulator introduction): see Section 4.7 of this course
  - ascending generalisation
  - tupling generalisation: replace a parameter by a list of parameters of the same type

  These standard methods aim at improving the time and/or space consumption of programs constructed without generalisation

Example: Analysing an Algorithm for the Towers of Hanoi

The end of the world, according to a Buddhist story ...

Initial state:



Rules:
Only move the top-most disk of a tower
Only move a disk onto a larger disk

Objective and final state: Move all the disks from tower `A` to tower `C`, without violating any rules

Problem: Write a program that determines a (minimal) sequence of movements to be done for reaching the final state from the initial state, without violating any rules

This is an example of a *planning problem*:
Artificial Intelligence

Specification

Movements

The movements are represented by a string of characters:

```
MOVE A B
MOVE A C
MOVE B C
```

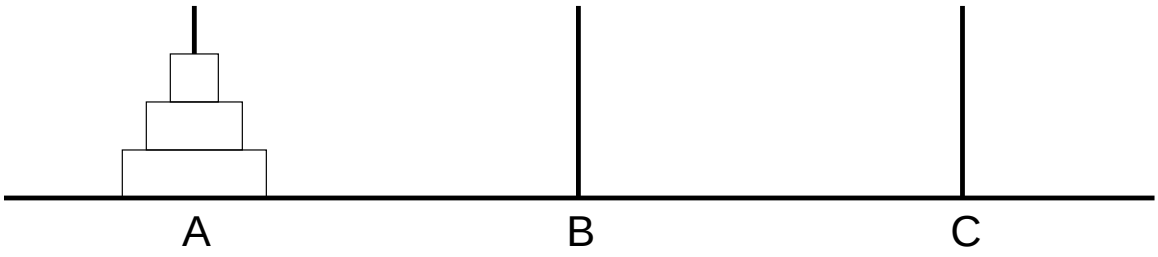
which shall be written in SML as:

```
"MOVE A B \n MOVE A C \n MOVE B C \n"
```

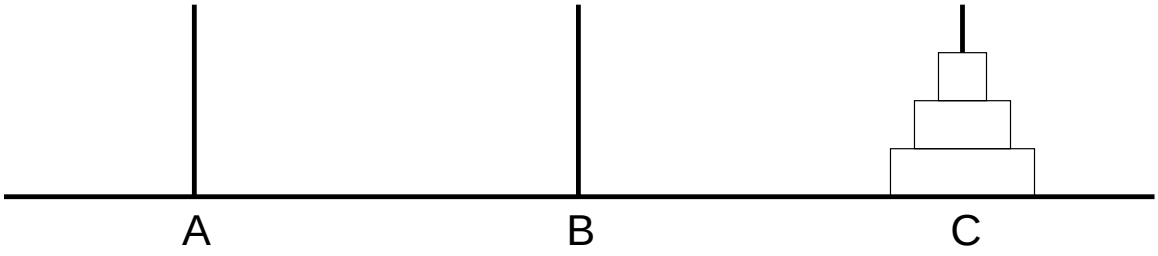
Specification

function hanoi n (start,aux,arrival)
TYPE: int $\longrightarrow$ (string $*$ string $*$ string) $\longrightarrow$ string
PRE: n $\geq$ 0
POST: description of the movements to be done
for transferring n disks from tower **start** to tower **arrival**,
using tower **aux**, without violating any rules

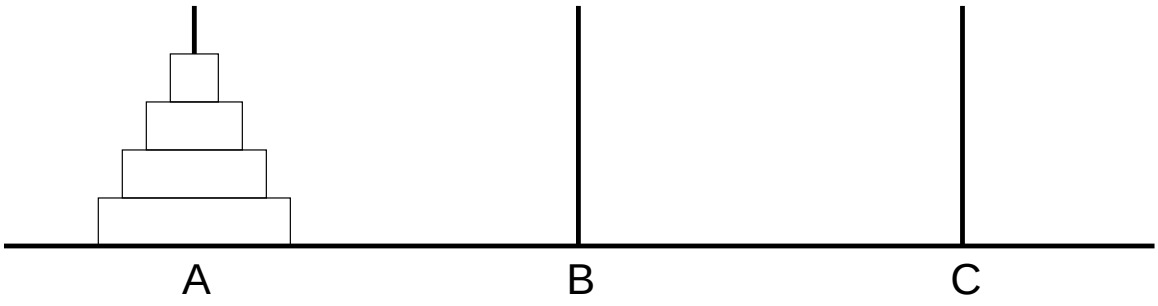
Example 1



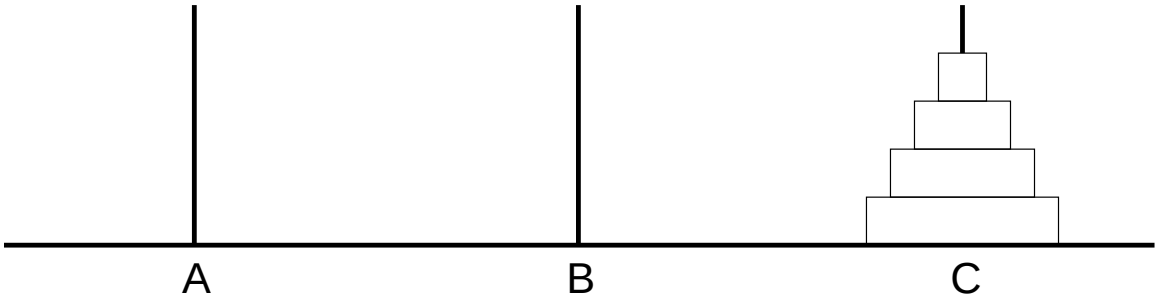

```
MOVE A C
MOVE A B
MOVE C B
MOVE A C
MOVE B A
MOVE B C
MOVE A C
```



Example 2

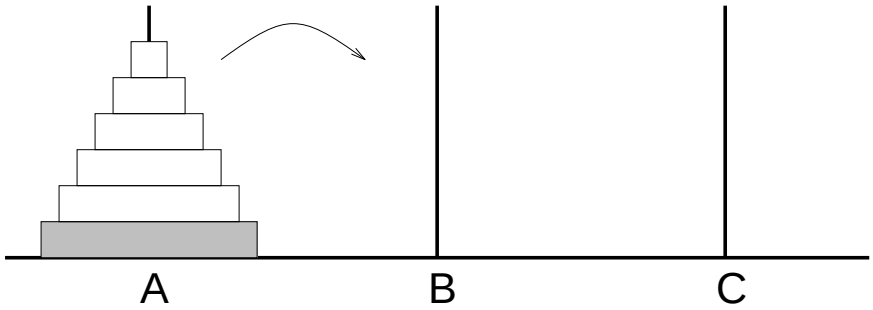



```
MOVE A B   MOVE A C
MOVE B C   MOVE A B
MOVE C A   MOVE C B
MOVE A B   MOVE A C
MOVE B C   MOVE B A
MOVE C A   MOVE B C
MOVE A B   MOVE A C
MOVE B C
```

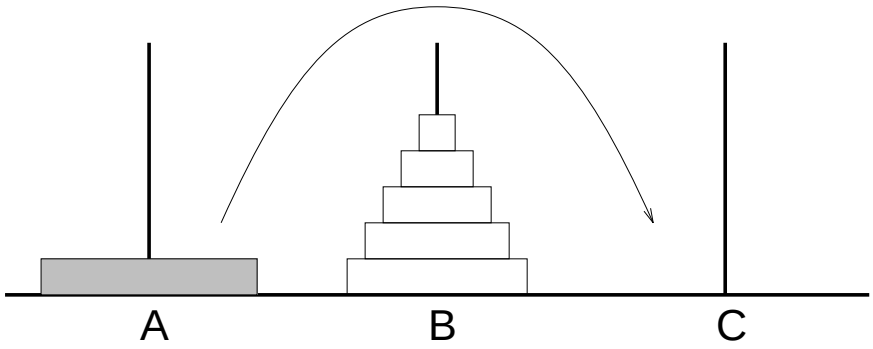


Strategy

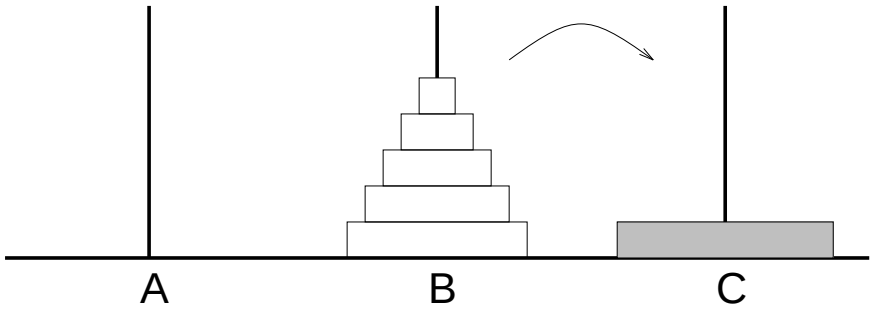
How to solve hanoi n ("A","B","C")?



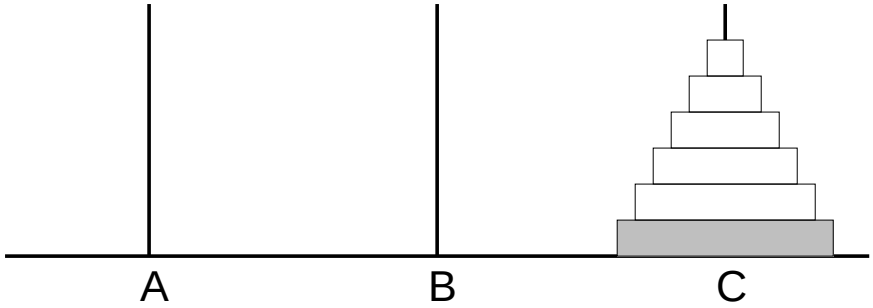
Movements to be done: hanoi (n-1) ("A","C","B")



Movement to be done: MOVE A C



Movements to be done: hanoi (n-1) ("B","A","C")



Construction of a program

Variant: n
Error case: $n < 0$: produce an error message
Base case: $n = 0$: no movement is needed
General case: $n > 0$: double recursive usage of hanoi with $n-1$

Program (`hanoi.sml`)

```
local
    fun hanoiAux 0 (start,aux,arrival) = ""
      | hanoiAux n (start,aux,arrival) =
          (hanoiAux (n−1) (start,arrival,aux))
          ^ "MOVE " ^ start ^ " " ^ arrival ^ "\n"
          ^ (hanoiAux (n−1) (aux,start,arrival))
in  fun hanoi n (start,aux,arrival) =
        if n < 0 then error "hanoi: negative number of disks"
        else hanoiAux n (start,aux,arrival)
end
```

Example

```
- print (hanoi 3 ("A","B","C")) ;
MOVE A C
MOVE A B
MOVE C B
MOVE A C
MOVE B A
MOVE B C
MOVE A C
```

Complexity

Will the end of the world be provoked soon
if we evaluate `hanoi 64 ("A","B","C")`,
even on the fastest computer of the year 2020?!

How many movements must be made for solving
the problem of the Towers of Hanoi with n disks?
Let $M(n)$ be this number of movements
The complexity of the function `hanoi` n is $\Theta(M(n))$

From the SML program, we get the *recurrence equations*:

$$M(0) = 0$$
$$M(n) = 2 \cdot M(n-1) + 1 \qquad \text{for } n > 0$$

How to solve these equations?

- Guess the result and prove it by induction
- Iterative method
- Application of a pre-established formula

Proof by induction

Theorem: $M(n) = 2^n - 1$

Base: $n = 0$

$$M(0) = 0 = 2^0 - 1$$

Induction: $n > 0$

$$\begin{aligned} M(n) &= 2 \cdot M(n-1) + 1 \\ &= 2 \cdot (2^{n-1} - 1) + 1 \\ &= 2^n - 1 \end{aligned}$$

Hence: the complexity of $\mathsf{hanoi}\, n$ is $\Theta(2^n)$

Iterative method

$$\begin{aligned} M(n) &= 2 \cdot M(n-1) + 1 \\ &= 2 \cdot (2 \cdot M(n-2) + 1) + 1 \\ &= 4 \cdot M(n-2) + 3 \\ &= 8 \cdot M(n-3) + 7 \\ &= 2^3 \cdot M(n-3) + (2^3 - 1) \\ &= \ldots \\ &= 2^k \cdot M(n-k) + (2^k - 1) \\ &= \ldots \\ &= 2^n \cdot M(0) + (2^n - 1) \\ &= 2^n - 1 \end{aligned}$$

Hence: the complexity of $\mathsf{hanoi}\, n$ is $\Theta(2^n)$

Application of a pre-established formula

General formula

$$C(n) = a \cdot C(n-1) + b$$

Normal form

$$C(n) = a^n \cdot C(0) + b \cdot \sum_{0 \le i < n} a^i$$

Particular cases

a = 1: $C(n) = C(0) + b \cdot n = \Theta(n)$

a = 2: $C(n) = 2^n \cdot C(0) + b \cdot (2^n - 1) = \Theta(2^n)$

a ≠ 1: $C(n) = a^n \cdot C(0) + b \cdot \frac{a^n - 1}{a - 1} = \Theta(a^n)$

Hence: the complexity of `hanoi` n is $\Theta(2^n)$