Notes 8.0: Functions, recursion

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2013 session 1

Generalities

A function provides a convenient way to encapsulate some computation.

Users of a properly designed function do not have to worry about its implementation: they only need to know *what* is done and they can ignore *how* the job is done.

A function is declared thanks to a function declaration or function prototype, defined thanks to a function definition, and called or summoned by other functions, possibly many times, and possibly by the function itself. Even short functions that are called once only by a single function can be welcome, in that they can result in clearer code.

A function prototype just provides some very general information about the structure of the function; a function definition determines what the function does, and how it does it.

Function definitions can appear in any order, and in one or several source files; still no function definition can be split between files.

An example

```
#include <stdio.h>
#include <stdlib.h>
int power(int, int);             // function prototype

int main(void) {
    for (int p = 0; p < 10; ++p) // call function twice
        printf("%d %d %d\n", p, power(2, p), power(-3, p));
    return EXIT_SUCCESS;
}

/* power(base, n) raises base to the power n, n >= 0 */
int power(int base, int n) {    // function definition
    int p = 1;
    for (int i = 1; i <= n; ++i)
        p *= base;
    return p;
}
```

Declarations

A function should be declared before it is used by other functions.

```
int power(int base, int n);
```

declares that `power()` is a function that has two parameters of type `int`, and that returns to the calling function a value of type `int`.

This line could be replaced by:

```
int power(int, int);
```

Naming the parameters is useful only if it helps documenting the function.

The function `power()` is called twice by `printf()`, which itself is called by `main()`.

Each call to `power()` passes two arguments to `power()`.

Since `power()` returns a value of type `int`, an expression like `power(2,i)` evaluates to a value of type `int`.

Local variables; return

The names used by `power()` for its parameters in its definition are local to that definition and are not visible outside.

Many functions can use local variables having the same name without conflict.

For instance, the variable `p` in the definition of `main()` is unrelated to the variable `p` in the definition of `power()`.

Any expression can follow `return`, provided that it evaluates to a value of type given by the function prototype.

A function definition should return a value of the type given by the function prototype, unless that function prototype uses `void` to indicate that the function does not return any value.

The calling function (or the operating system) can ignore the value returned by a called function.

Modifying a parameter

Similarly to any other variable, the value of a parameter can be modified in the body of a function, like in the following alternative definition of the `power()` function.

```
int power(int base, int n) {
    int p = 1;
    for ( ; n; --n)
        p *= base;
    return p;
}
```

Both definitions of the function would not behave nicely if called with a second argument whose value is negative. A more robust implementation would first check that the second argument is positive, but this raises the problem of what value the function should return.

Call by value (1)

In C all function arguments are passed by value. This means that the called function is given a *copy* of the arguments, whose values are stored in temporary variables, rather than the original variables.

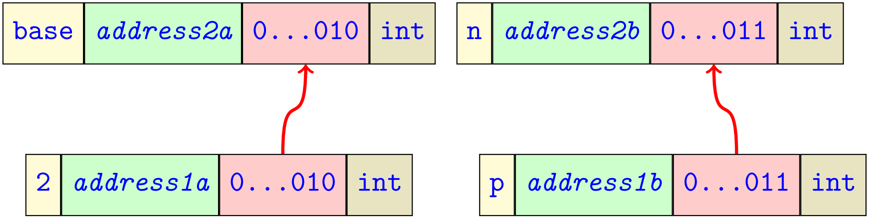
Hence a called function cannot directly alter the value of a variable that the calling function provides as an argument to the called function.

In the previous program, the call to `power(2, p)` does not change the value of `p`:

- for the second implementation of `power()`, the parameter `n` is a new, temporary variable, that is counted down until it becomes zero when `power(2, p)` returns;
- when control returns to the calling function `printf()`, the value of `p`, which was provided as an argument to `power()`, is still the same as before the call.

Call by value (2)

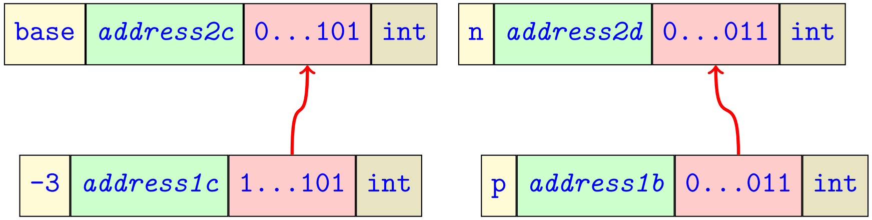
When `power(2, p)` is called with `p` holding the value `3`, data items of name `base` and `n` are created. . .



. . . and have their value assigned from the data items of name `2` and `p` in the calling function.

Call by value (3)

When `power(-3, p)` is called with `p` holding the value `3`, new data items of name `base` and `n` are created. . .



. . . and have their value assigned from the data items of name `-3` and `p` in the calling function. The names are the same (`base` and `n`), but the addresses *might* be different (*`adress2a`* and *`address2b`* versus *`adress2c`* and *`address2d`*), but do not *have* to be different—provided that one call to `power()` has been completed when the other call begins.

The possible effects of a function call

Some functions cause action to take place: *e.g.*, `printf()` causes data to be printed on standard output.

Functions can also provide values in 3 different ways:

- directly by returning the value of a local variable of a parameter;
- indirectly by modifying the value of a global variable (that can be accessed by many functions),
- indirectly, by modifying the value of a variable whose address is passed to the function by the calling function.

Functions and program modularity

Functions make a program more modular, easier to modify and fix.

For instance, to read in a list of numbers, sort the numbers, find their average and print a bar graph, it is natural to organize the program as:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100      // max number of inputs

int main(void) {
    double list[SIZE];
    read_numbers(list, SIZE);
    sort(list, SIZE);
    average(list, SIZE);
    bargraph(list, SIZE);
    return EXIT_SUCCESS;
}
```

No argument, no returned value (1)

Consider the following program.

```
#include <stdio.h>
#include <stdlib.h>
#define NAME "CSE School office"
#define ADDRESS "UNSW Sydney NSW 2052"
#define NB_OF_STARS 45
void starbar(void);

int main(void) {
    starbar();
    printf("%s\n", NAME);
    printf("%s\n", ADDRESS);
    starbar();
    return EXIT_SUCCESS;
}
...
```

No argument, no returned value (2)

```
...
void starbar(void) {
    for (int count = 0; count < NB_OF_STARS; ++count)
        printf("*");
    printf("\n");
}
```

The function prototype for `starbar()`, in accordance with the definition of `starbar()`, tells the compiler how the function is meant to be used:

- `void` before the function name indicates that the function type is void: it returns no value because it does not need to provide any information to the calling function.
- `void` between the parentheses after the function name indicates that the function takes no argument: it has no input because it needs no information from the calling function.

Some arguments, no returned value (1)

The function show_n_char() below is declared to be of type void, and has two formal parameters of respective type char and int.

```
#include <stdio.h>
#include <string.h>    // for the function strlen()
#define NAME "CSE School office"
#define ADDRESS "UNSW Sydney NSW 2052"
#define NB_OF_STARS 45
#define SPACE ' '

void show_n_char(char, int);

void show_n_char(char c, int nb_of_chars) {
    for (int i = 0; i < nb_of_chars; ++i)
        printf("%c", c);
}
```

Some arguments, no returned value (2)

```
int main(void) {
    show_n_char('*', NB_OF_STARS);  // a constant as argument
    printf("\n");
    int spaces = (NB_OF_STARS - strlen(NAME)) / 2;
    show_n_char(SPACE, spaces);     // a variable as argument
    printf("%s\n", NAME);
    show_n_char(SPACE, (NB_OF_STARS - strlen(ADDRESS)) / 2);
                                    // an expression as argument
    printf("%s\n", ADDRESS);
    show_n_char('*', NB_OF_STARS);
    printf("\n");
    return 0;
}
```

`main()` calls `show_n_char()` four times by providing actual arguments, assigned to the corresponding formal parameters.

Some arguments, a returned value (1)

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

int min1(int, int);
int min2(int, int);
int min3(int, int);

int main(void) {
    int a, b;
    p_prompt("Enter pair of integers: ", "%d %d", &a, &b);
    printf("Thrice the min of %d and %d: %d %d %d.\n",
           a, b, min1(a, b), min2(a, b), min3(a, b));
    return EXIT_SUCCESS;
}
```

Some arguments, a returned value (2)

```
int min1(int n, int m) {
    if (n < m)
        int min = n;
    else
        int min = m;
    return min;
}
int min2(int n, int m) {
    return n < m ? n : m;
}
int min3(int n, int m) {
    if (n < m)
        return n;
    return m;
}
```

A few remarks

For functions of type `void`, a simple `return;` statement can be used, causing the function to terminate and return control to the calling function.

To use a function correctly, a program needs to know the function type before the function is used for the first time, which can be done

- by placing the function definition ahead of its first use; still this can make the program hard to read and gives less flexibility, or
- by placing all function prototypes ahead of the definitions of the functions—this is the preferred way—, or
- by placing the needed function prototypes in the function definitions. writing for instance

```
int main(void) {
    int min1(int, int);
    int min2(int, int);
    int min3(int, int);
```

More marbles: the need for recursion

The program tower_and_m_glass_marbles.c solves the following generalisation of the problem we have solved in the case where $m = 2$.

> *The user is prompted to enter the number of floors of a building. Using m marbles, one has to discover the highest floor, if any, such that dropping a marble from that floor makes it break, using a strategy that minimises the number of drops in the worst case (it is assumed that any marble would break when dropped from a floor when one marble breaks, and also when dropped from any higher floor; the marbles might not break when dropped from any floor).*

It uses a nontrivial form of recursion.

Recursion: generalities

Recursion is the process of one function calling itself.

To avoid that the function calls itself infinitely often:

- one or more base cases have to be provided;
- different parameters should be used from one call to the next.

Recursion can be replaced by loop constructions, but:

- recursion often offers a simpler solution to the problem to be solved;
- recursion is often more elegant, resulting in code that is more compact.

Recursive solutions are often less efficient, and sometimes dramatically so, than nonrecursive solutions.

Illustration

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

void up_and_down(int);
int main(void) {
    up_and_down(1);
    return EXIT_SUCCESS;
}

void up_and_down(int n) {
    printf("Level %d\n", n);    // print #1
    if (n < 4)
        up_and_down(n + 1);
    printf("LEVEL %d\n", n);    // print #2
}
```

Key observations

- Each level of function call produces its own *own* data items:

variables	n	n	n	n
after level 1 call	1			
after level 2 call	1	2		
after level 3 call	1	2	3	
after level 4 call	1	2	3	4
after return from level 4 call	1	2	3	
after return from level 3 call	1	2		
after return from level 2 call	1			
after return from level 1 call				

- When program flow completes one level of recursion, the program moves back to the previous level of recursion.
  - Statements that come *before* a recursive call are executed in the order from which the functions are called.
  - Statements that come *after* a recursive call are executed in the opposite order from which the functions are called.

One problem, many solutions (1)

```
#include <stdio.h>
int iter_sum(int, int);
int rec_sum1(int, int);
int rec_sum2(int, int);
int rec_sum3(int, int);

int main(void) {
    printf("Iterative solution:
                 %d\n", iter_sum(5, 12));
    printf("First recursive solution:
                 %d\n", rec_sum1(5, 12));
    printf("Second recursive solution:
                 %d\n", rec_sum2(5, 12));
    printf("Third recursive solution:
                 %d\n", rec_sum3(5, 12));
    return 0;
}
```

One problem, many solutions (2)

```
int iter_sum(int m, int n) {
    int sum = 0;
    for (int i = m; i <= n; ++i)
        sum += i * i;
    return sum;
}

int rec_sum1(int m, int n) {
    if (m < n)
        return  m * m + rec_sum1(m + 1, n);
    return m * m;
}
```

One problem, many solutions (3)

```
int rec_sum2(int m, int n) {
    if  (m < n)
        return rec_sum2(m, n - 1) + n * n;
    return n * n;
}

int rec_sum3(int m, int n) {
    if (m == n)
        return m * m;
    int middle = (m + n) / 2;
    return rec_sum3(m, middle) + rec_sum3(middle + 1, n);
    }
}
```

Iteration versus recursion (1)

The Fibonacci sequence starts with 0 and 1; any other member of the sequence is the sum of the previous two members:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad \ldots$$

```
#include <stdio.h>
#include <p_io.h>

long long iter_fibonacci(int);
long long rec_fibonacci(int);

int main(void) {
    int num;
    p_prompt("Enter a positive integer: ", "%>=0d", &num);
    printf("Via iteration: %lld\n", iter_fibonacci(num));
    printf("Via recursion: %lld\n", rec_fibonacci(num));
    return 0;
}
```

Iteration versus recursion (2)

```
long long iter_fibonacci(int n) {
    int previous = 0, current = 1;
    if (n < 2)
        return n;
    for (int i = 2; i <= n; ++i) {
        int temp = current;
        current += previous;
        previous = temp;
    }
    return current;
}

long long rec_fibonacci(int n) {
    if (n >= 2)
        return rec_fibonacci(n - 2) + rec_fibonacci(n - 1);
    return n;
}
```

Tail recursion

The problem with the recursive computation of the Fibonacci numbers is that `rec_fibo()` makes calls that force the same computation to be done repeatedly instead of only once.

In the simpler form of recursion, there is only one call, and that call is at the end of the function (or just before the `return;` statement for functions whose type is not `void`); this is called tail recursion.

Tail recursion acts as a loop. Most compilers optimize the code and translate tail recursion into a loop construct. Indeed the recursive version is less efficient because:

- each recursive call gets its own set of variables (data items) that it places on the stack, which uses more memory;
- function calls take time.

The program conversion.c converts a number from base 10 to base 2 using two implementations one of which is tail recursive.

A famous example: the towers of Hanoi (1)

The internet offers many ways to play the game, thanks to applets that you can sometimes install on your own machine; see for instance http://www.mathcs.org/java/programs/Hanoi/index.html.

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

void move_towers(int, int, int, int);

int main(void) {
    int num;
    p_prompt("Enter a positive integer: ", "%>=0d", &num);
    move_towers(num, 1, 3, 2);
    return EXIT_SUCCESS;
}
```

A famous example: the towers of Hanoi (2)

```
/* Move a tower of n disks on the start-peg
   to the finish-peg using the spare-peg
   as an intermediary.*/
void move_towers(int n, int start, int finish, int spare) {
    if (n == 1)
        printf("Move a disk from peg %1d to peg %1d\n",
                                  start, finish);
    else {
        move_towers(n - 1, start, spare, finish);
        printf("Move a disk from peg %1d to peg %1d\n",
                                  start, finish);
        move_towers(n - 1, spare, finish, start);
    }
}
```