
Computer Programming: Skills & Concepts (CP1)

Programming Techniques: Flags and Recursion

John Longley

18th November 2008



Flags

A *flag* is simply a Boolean variable, often used to record whether some significant event has yet happened.

Example: does a word contain the letter c?

```
#define FOUND 1
#define NOT_FOUND 0
char word[20] = "abracadabra!";

int flag = NOT_FOUND;
for (int i=0; word[i] != '\0'; i++) {
    if (word[i] == 'c')
        flag = FOUND;
}
```

Flags vs. exiting

That was a rather silly example, because we could have written:

```
int Contains_c () {
    for (int i=0; word[i] != '\0'; i++) {
        if (word[i] == 'c')
            return FOUND;
    }
    return NOT_FOUND;
}
```

However, this trick of exiting at different points isn't always possible . . .

Multiple flags

```
int Contains_c_and_q ()
    int flag_c = NOT_FOUND;
    int flag_q = NOT_FOUND;
    for(int i=0; word[i] != '\0'; i++) {
        if (word[i] == 'c')
            flag_c = FOUND;
        else if (word[i] == 'q')
            flag_q = FOUND;
    }
    return (flag_c && flag_q)
}
```

Another example: bubble sort

Idea: if we didn't do any swaps on the previous pass, we've finished.

```
int changed = TRUE;
while (changed) do {
    changed = FALSE;
    for (j = 0; j < n; ++j) {
        if (a[j] > a[j+1]) {
            Swap (&a[j], &a[j+1]);
            changed = TRUE;
        }
    }
}
```

Introduction to recursion

Task: write a function that computes factorial

$$n! = \begin{cases} n \times (n - 1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Could use a for loop (nothing new here):

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

Factorial using recursion

Another method:

```
int factorial (int n) {  
    if (n<=1)  
        return 1;  
    return n * factorial(n-1);  
}
```

The function `factorial` calls itself!

Execution of recursion

```
factorial(5)
  return 5 * factorial(4);
    return 4 * factorial(3);
      return 3 * factorial(2);
        return 2 * factorial(1);
          return 1;
            return 2 * 1;
              return 3 * 2
                return 4 * 6
                  return 5 * 24
120
```

Another example: Fibonacci numbers

The Fibonacci numbers are the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

$\frac{F(n+1)}{F(n)}$ converges to the **golden ratio** 1.6180339...

Recursive computation of Fibonacci numbers

```
int fibonacci (int n) {  
    if (n==0)  
        return 0;  
    if (n==1)  
        return 1;  
    return fibonacci (n-1) + fibonacci (n-2);  
}
```

- Very inefficient: runtime is exponential in n , owing to vast amounts of duplicated computation.
- Can be improved: e.g. write a recursive function that returns a *pair* of Fibonacci numbers $(F(n), F(n - 1))$.

A realistic example: Quicksort

The main function: sorts the portion of some array from left to right.

```
void quicksort (int *left, int *right) {
    int pivot, *pos ;
    if (find_pivot (left, right, &pivot)) {
        /* NB returns false when *left==*right */
        p = partition (left, right, pivot) ;
        /* partition does the hard work */
        quicksort (left, p-1) ;
        quicksort (p, right) ;
    }
}
```

Iteration vs. recursion

- Recursion and iteration perform similar tasks: “looping”.
- Recursive code is often shorter and cleaner than iterative code.
- Iteration is typically more efficient (in terms of time and space), though often this doesn't matter.
- Recursive programs involve a “mental twist” which can make them harder to understand at first.
- Always remember to ensure that your recursion “bottoms out” in a base case.