Fundamentals of Computer Programming

C Programming

6. Functions

Contents

- The General Form of a Function
- Understanding the Scope of a Function
- Function Arguments
- argc and argv— Arguments to main( )
- The return Statement
- What Does main( ) Return?
- Recursion
- Function Prototypes

The General Form of a Function

```
ret-type function-name(parameter list)
{
   body of the function
}
```

- The `ret-type` specifies the type of data that the function returns.
- A function may return any type of data except an array.
- The `parameter list` is a comma-separated list of variable names and their associated types.
- The parameters receive the values of the arguments when the function is called.

[2] p. 148

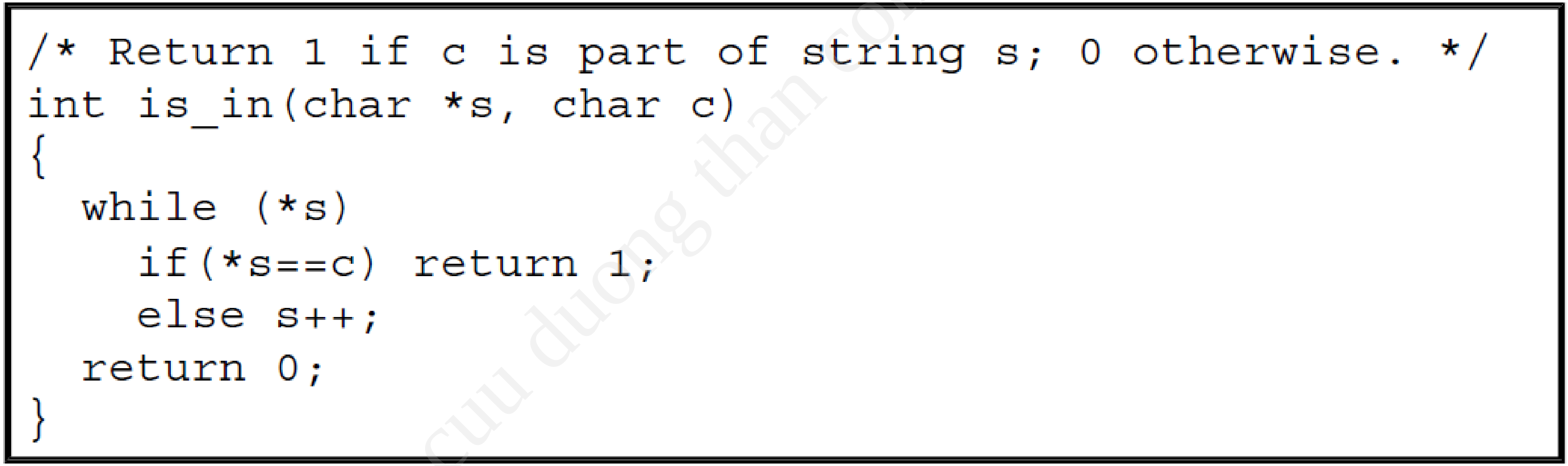
Understanding the Scope of a Function

- A function defines a block scope.
- A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function.
- Variables that are defined within a function are local variables.
- A local variable comes into existence when the function is entered and is destroyed upon exit.
- The formal parameters to a function also fall within the function's scope.
- All functions have file scope. Thus, you cannot define a function within a function.

[2] pp. 148-149

Function Arguments

- If a function is to accept arguments, it must declare the parameters that will receive the values of the arguments.



```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
  while (*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

- The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character **c** is part of the string **s**; otherwise, it returns 0.

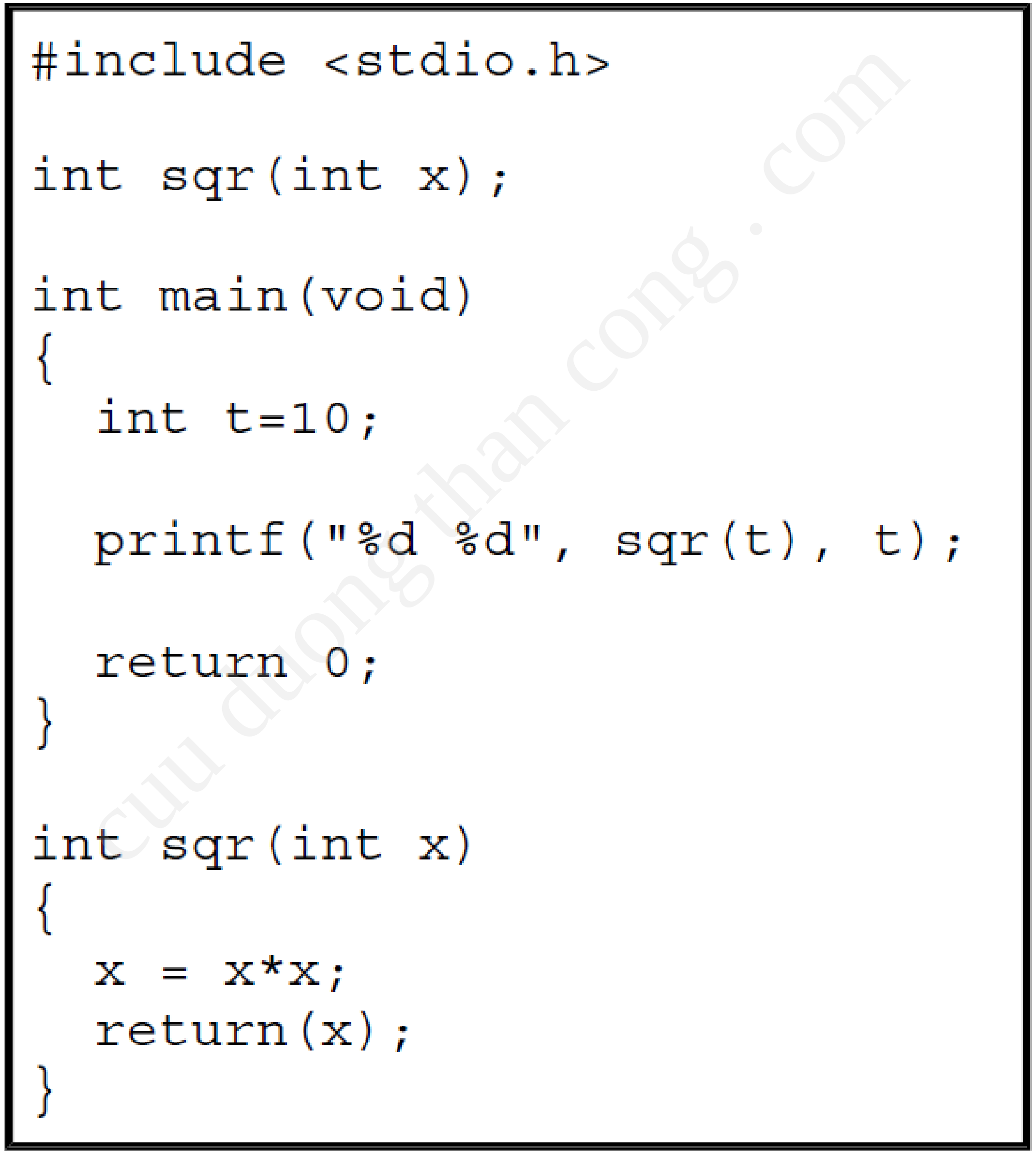
[2] p. 149

Call by Value, Call by Reference

- There are two ways that arguments can be passed to a subroutine:
  - *call by value*: copies the *value of* an argument into the formal parameter of the subroutine.
  - *call by reference*: the address is used to access the actual argument used in the call.
- With few exceptions, C uses *call by value* to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

[2] pp. 149-150

Call by Value

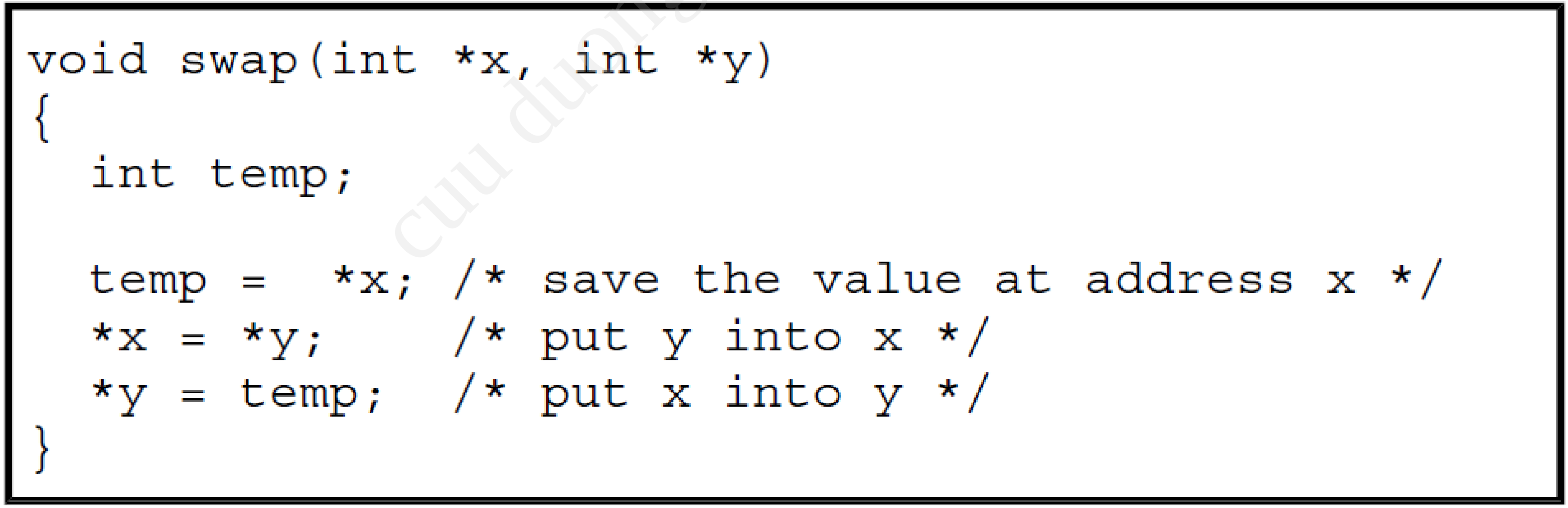


```
#include <stdio.h>

int sqr(int x);

int main(void)
{
  int t=10;

  printf("%d %d", sqr(t), t);

  return 0;
}

int sqr(int x)
{
  x = x*x;
  return(x);
}
```

[2] p. 150

Creating a Call by Reference

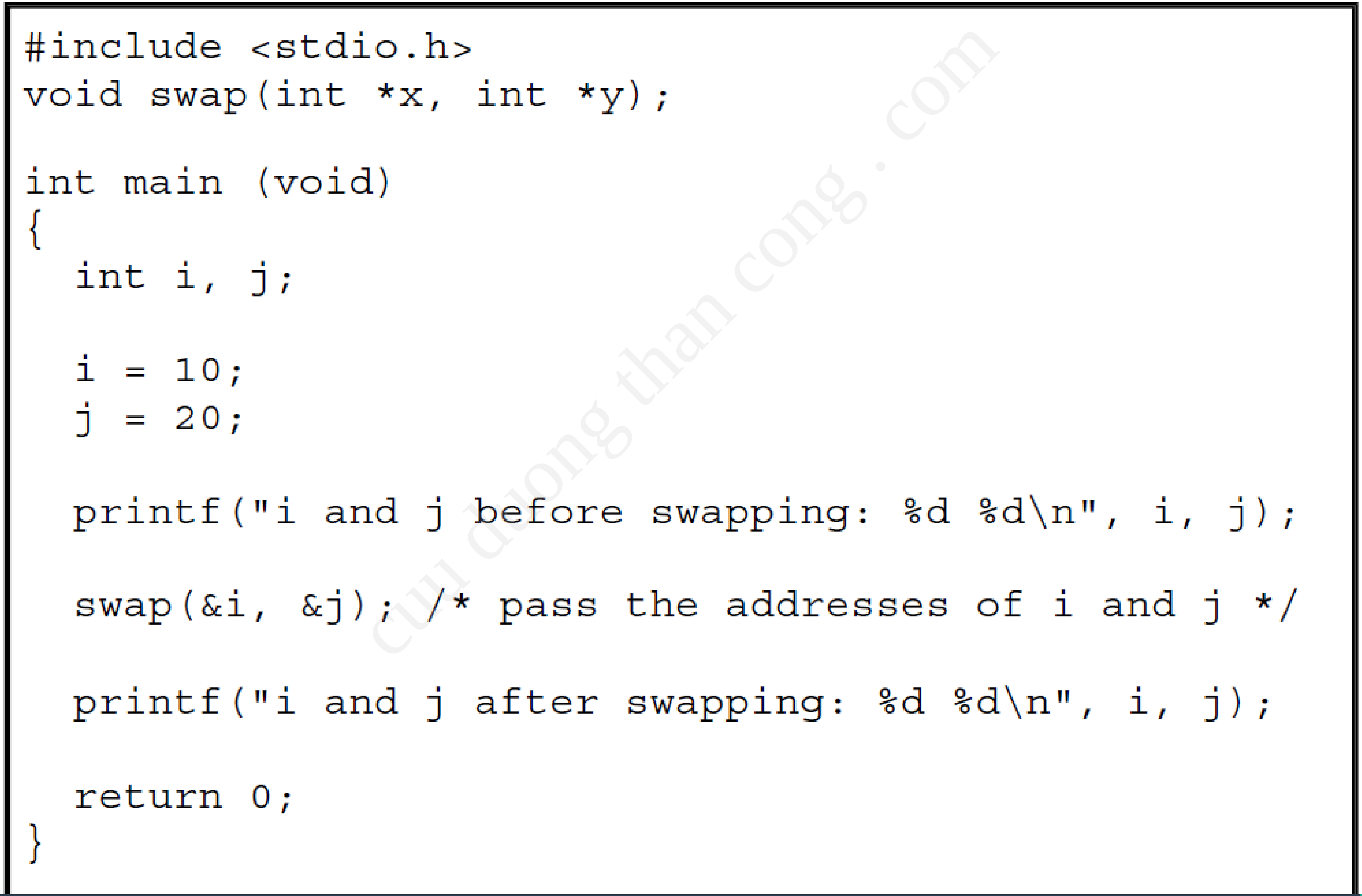
- You can create a *call by reference* by passing *a pointer to an argument*, instead of passing the argument itself.
- Since *the address of the argument* is passed to the function, code within the function *can change the value* of the argument outside the function.



```
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 */
}
```

[2] pp. 150-151

Addresses of the Arguments



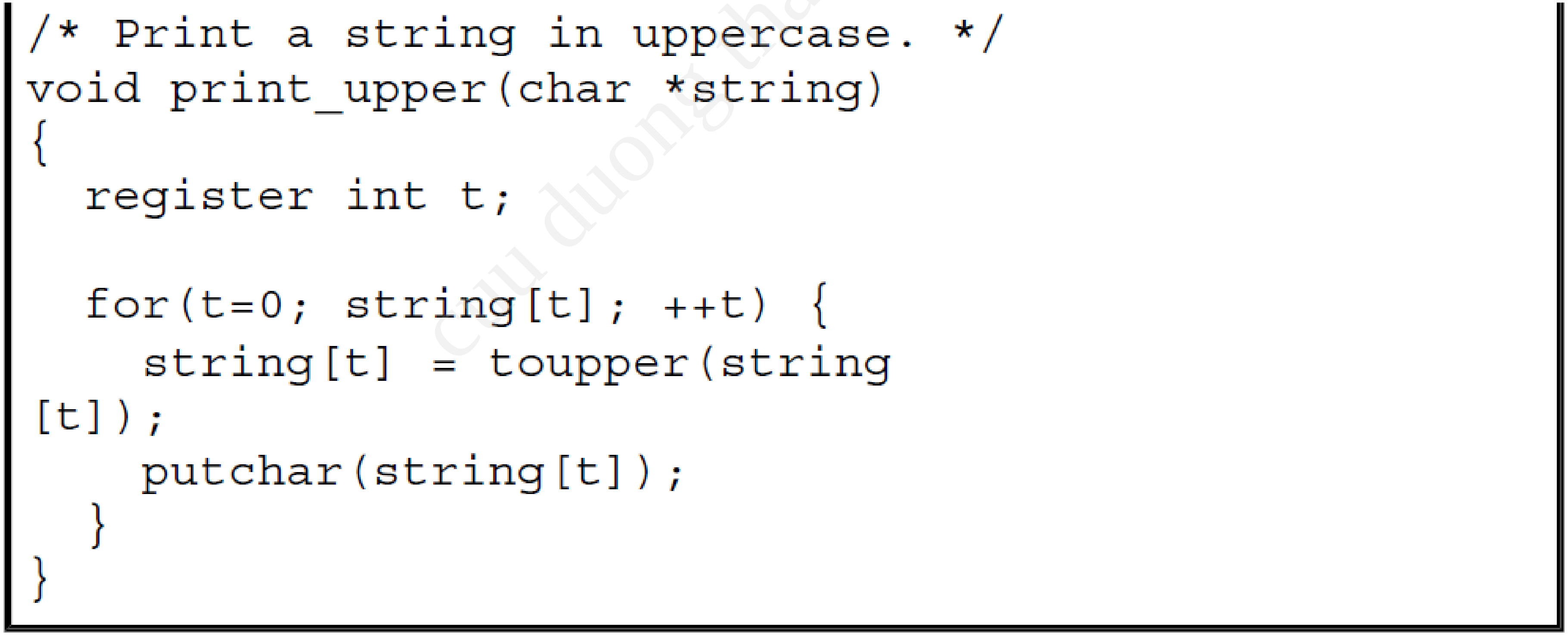
```
#include <stdio.h>
void swap(int *x, int *y);

int main (void)
{
  int i, j;

  i = 10;
  j = 20;

  printf("i and j before swapping: %d %d\n", i, j);

  swap(&i, &j); /* pass the addresses of i and j */

  printf("i and j after swapping: %d %d\n", i, j);

  return 0;
}
```

[2] p. 151

Calling Functions with Arrays

- When an array is used as a function argument, *its address* is passed to a function.

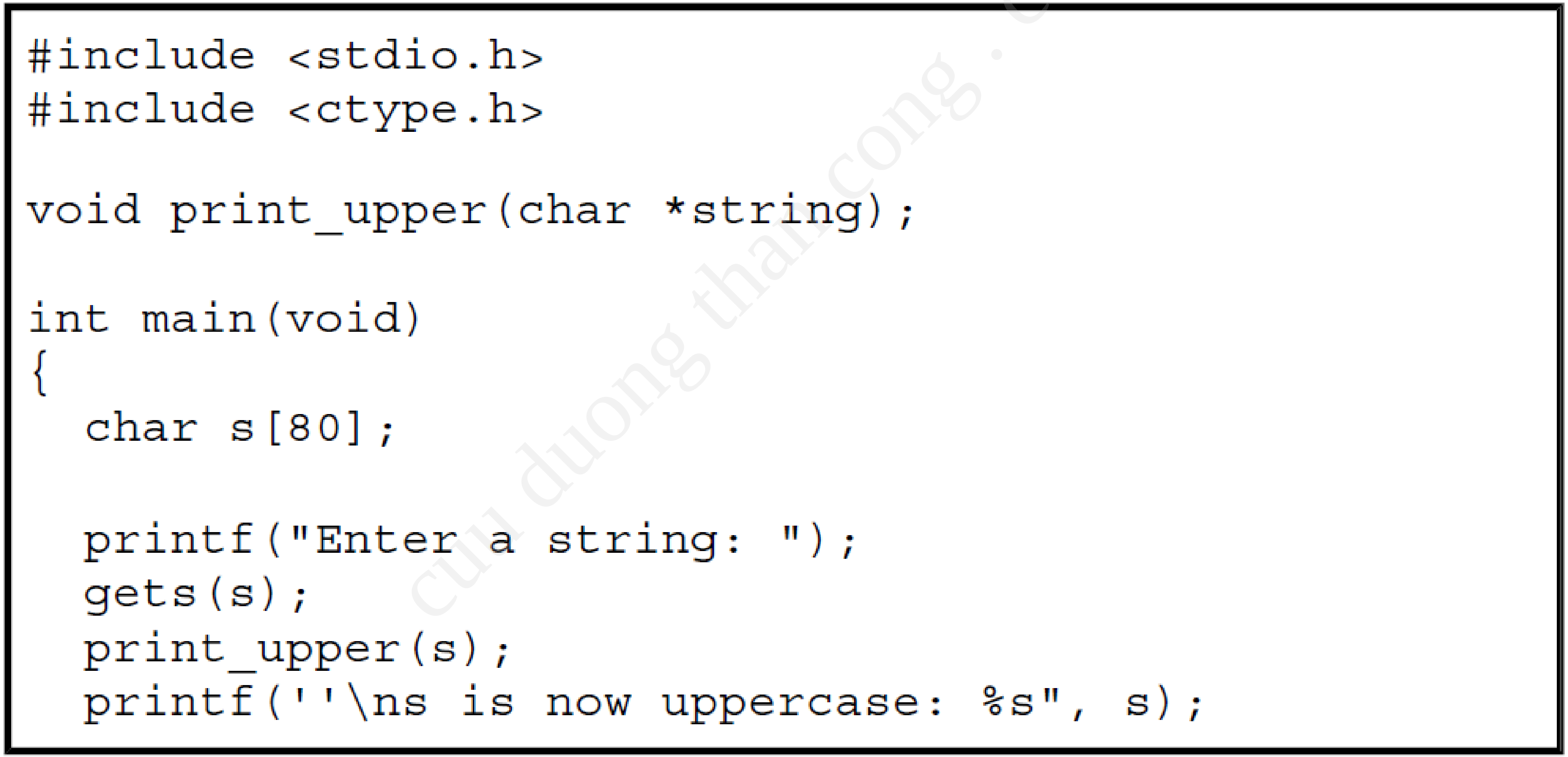
- The code inside the function is operating on, and potentially *altering*, the actual contents of the array used to call the



```
/* Print a string in uppercase. */
void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t) {
    string[t] = toupper(string
[t]);
    putchar(string[t]);
  }
}
```

[2] pp. 152-153

print_upper( )



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

void print_upper(char *string);

int main(void)
{
  char s[80];

  printf("Enter a string: ");
  gets(s);
  print_upper(s);
  printf(''\ns is now uppercase: %s", s);
```

[2] p. 152

argc and argv— Arguments to main( )

- Ref: [2] pp. 155-158

The return Statement

- Two important uses:
  - It causes an *immediate exit from the function*. That is, it causes program execution to return to the calling code.
  - It can be used to *return a value*.

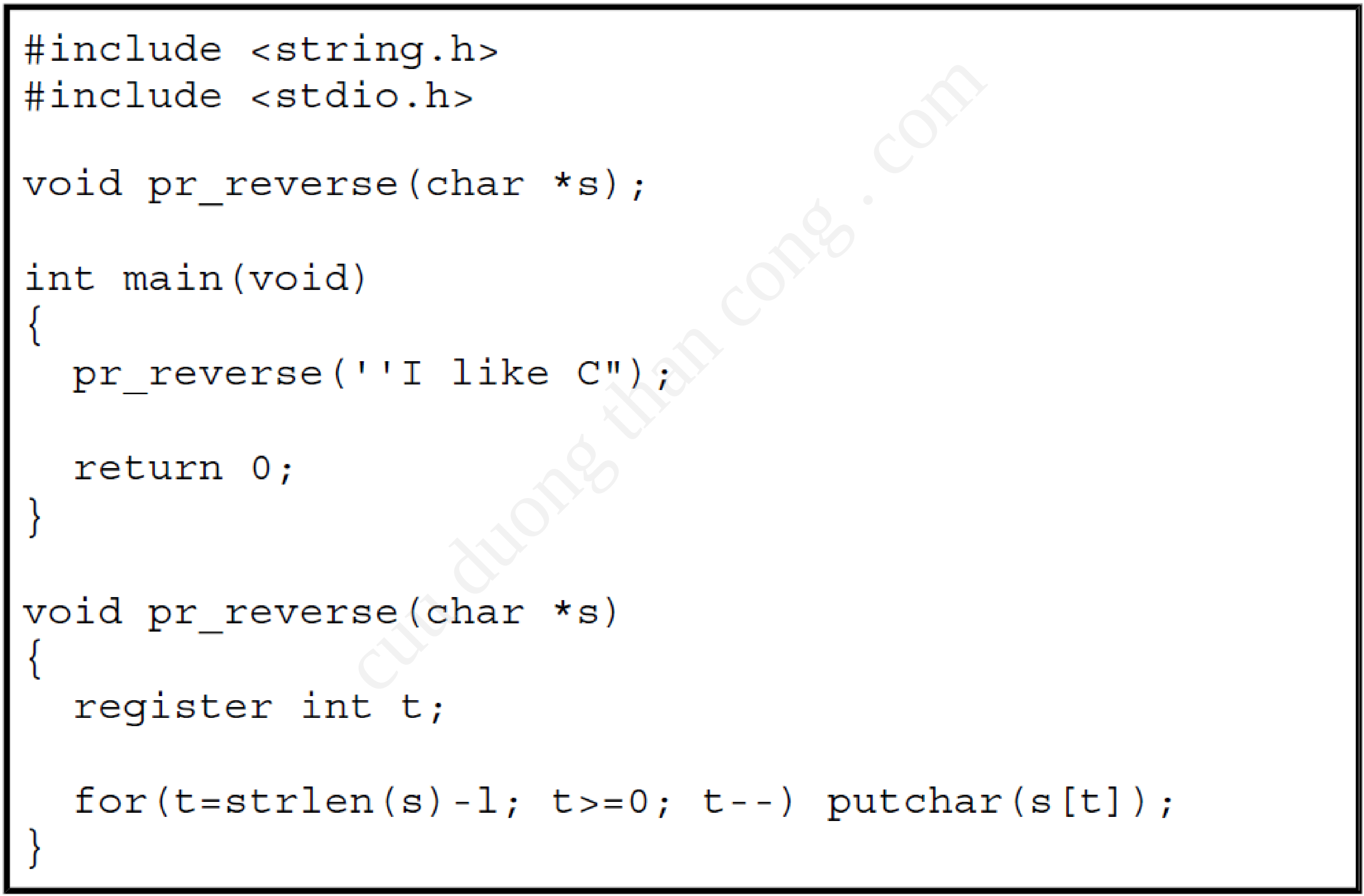
[2] p. 158

Returning from a Function

- A function terminates execution and returns to the caller in two ways.
- The first occurs when the *last* statement in the function has executed.
- Most functions rely on the **return** statement to stop execution either because *a value must be returned* or to make a function's code simpler and more efficient.
- A function may contain several **return** statements.

[2] pp. 158-159

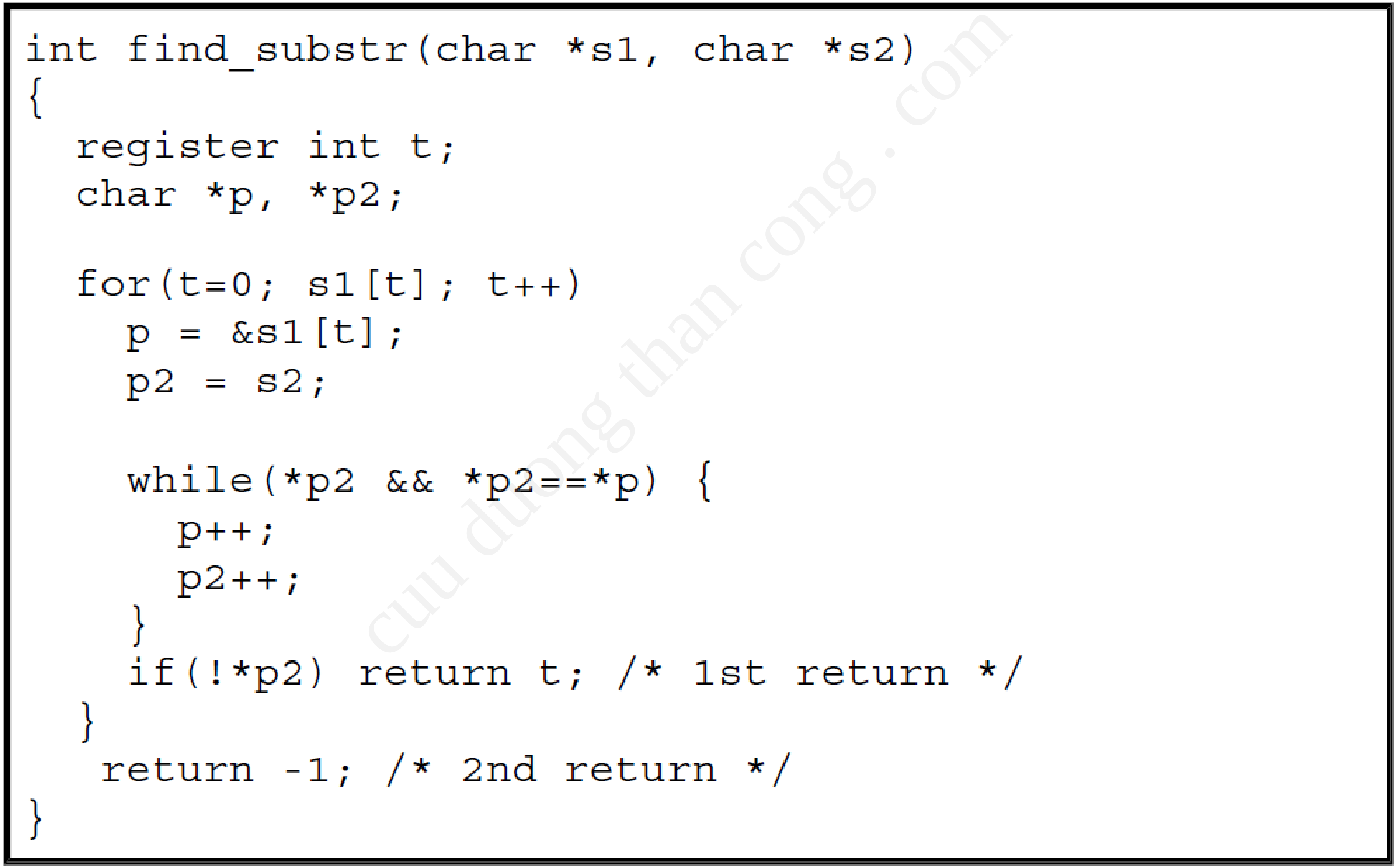
Returning from a Function – Default Method



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

void pr_reverse(char *s);

int main(void)
{
  pr_reverse(''I like C");

  return 0;
}

void pr_reverse(char *s)
{
  register int t;

  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

[2] p. 159

Returning from a Function – Return a Value

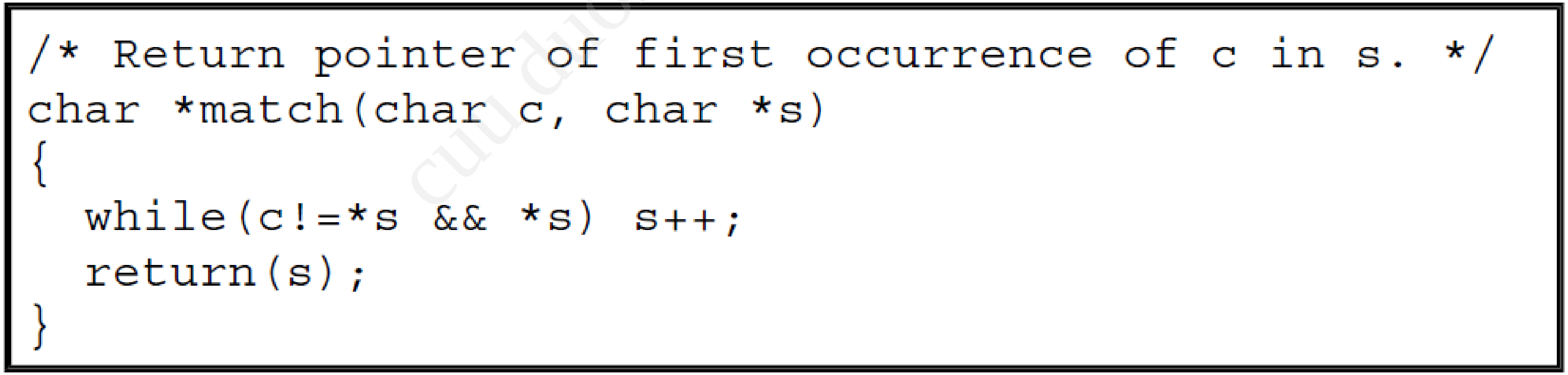


```
int find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;

  for(t=0; s1[t]; t++)
    p = &s1[t];
    p2 = s2;

    while(*p2 && *p2==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* 1st return */
  }
   return -1; /* 2nd return */
}
```

[2] p. 160

Returning Pointers

- To return a pointer, a function must be declared as having a *pointer return type*.
- For example, the following function returns *a pointer* to the first occurrence of the character c in string s: If no match is found, a pointer to the null terminator is returned.

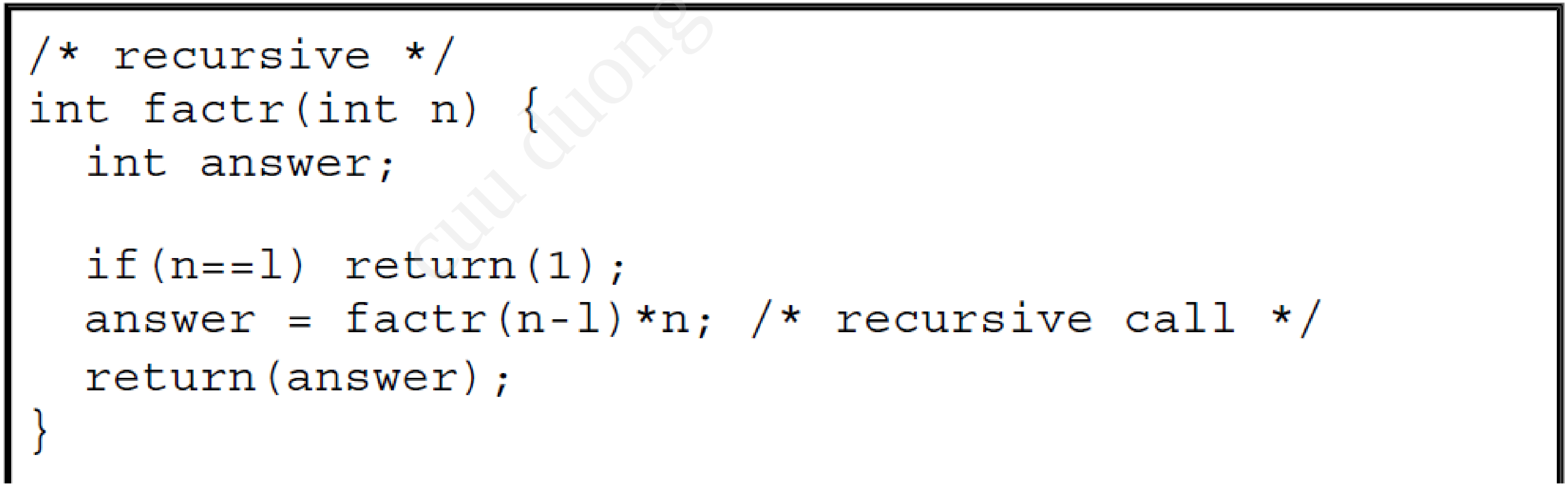


```
/* Return pointer of first occurrence of c in s. */
char *match(char c, char *s)
{
  while(c!=*s && *s) s++;
  return(s);
}
```

[2] pp. 162-163

Recursion

- In C, a function can call itself. In this case, the function is said to be *recursive*.
- When a function calls itself, *a new set of local variables and parameters* are allocated storage on the stack, and the function code is executed from the top with these new variables.



```
/* recursive */
int factr(int n) {
  int answer;

  if(n==1) return(1);
  answer = factr(n-1)*n; /* recursive call */
  return(answer);
}
```

[2] pp. 164-165