

---

# Computer Programming: Skills & Concepts (CP1)

## Parameters, & and \*

Aris Efthymiou

16th October, 2008



## A footnote to the previous lecture

In general, a function must be declared (i.e., defined) before use. So, the level of abstraction will generally increase from the lowest-level functions at the beginning of the program, to the highest-level at the end, finishing with `main` itself.

If you need or want to structure the program differently, there is a way out. The compiler only needs to see the *header* of the function before the function is used: the body of the function can come later.

In this situation, the disembodied header is called a function *prototype*. This style of programming is widespread in Kelley and Pohl.

## An example

```
#include <stdlib.h>

int succ(int n);

int main(void) {
    printf("The successor of 3 is %d.\n", succ(3));
    return EXIT_SUCCESS;
}

int succ(int n) {
    return ++n;
}
```

## Identifiers are optional in prototypes

```
#include <stdlib.h>

int succ(int);

int main(void) {
    printf("The successor of 3 is %d.\n", succ(3));
    return EXIT_SUCCESS;
}

int succ(int n) {
    return ++n;
}
```

## A closer look at parameters

When a function is called, a correspondence is made between the actual and formal parameters.

There are a number of possible mechanisms for realising this correspondence. Some programming languages offer more than one such mechanism. In C, there is just one: *call by value*.

We consider the case of a function with one parameter, since functions with several parameters introduce no new issues.

## Call by value

The actual parameter is an expression of a certain type. The formal parameter is a variable of the same type.

- The actual parameter is evaluated to yield a value of the specified type.
- The formal parameter is initialised to that value. (Recall that the formal parameter is a particular kind of local variable.)
- The function body is executed.
- When `return` is reached (or the end of the function body) control passes to the point immediately after the function call.

## An example

```
int i = 3;

int succ(int n) {
    ++n;
    printf("Hi from \"succ\"!  The value of i is %d.\n", i);
    return n;
}

int main(void) {
    printf("The successor of %d is %d.\n", i, succ(i));
    printf("Hi from \"main\"!  The value of i is %d.\n", i);
    return EXIT_SUCCESS;
}
```

## Another example

```
void swap(int a, int b) {  
    a = b;  
    b = a;  
}
```

```
int main(void) {  
    int i = 1, j = 2;  
    printf("Checkpoint A:  i = %d and j = %d.\n", i, j);  
    swap(i, j);  
    printf("Checkpoint B:  i = %d and j = %d.\n", i, j);  
    return EXIT_SUCCESS;  
}
```

## Attempt 2

```
void swap(int *a, int *b) {  
    *a = *b;  
    *b = *a;  
}
```

```
int main(void) {  
    int i = 1, j = 2;  
    printf("Checkpoint A:  i = %d and j = %d.\n", i, j);  
    swap(&i, &j);  
    printf("Checkpoint B:  i = %d and j = %d.\n", i, j);  
    return EXIT_SUCCESS;  
}
```

## Explanation

- `&i` is the *address* of `i`. Instead of passing the value contained in `i` we pass a handle that allows the function `swap` to manipulate the variable `i`
- `int *a` declares `a` to be a *pointer* to an `int`. Initially `a` is set to point to `i`. `*a` denotes the integer variable pointed to by `a`. When we assign a value to `*a`, it is exactly as if we had assigned that value to `i` itself!

Using the combination of `&` and `*` we achieve the effect of *call by reference*.

## Attempt 3

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
int main(void) {  
    int i = 1, j = 2;  
    printf("Checkpoint A:  i = %d and j = %d.\n", i, j);  
    swap(&i, &j);  
    printf("Checkpoint B:  i = %d and j = %d.\n", i, j);  
    return EXIT_SUCCESS;  
}
```

## An example: ReadNumber from Practical 2

```
/*  
 * Read a number from the input stream.  
 *  
 * value: On success, value receives the value read.  
 *  
 * Return - TRUE if successful, FALSE otherwise.  
 */  
  
int ReadNumber(int *value);
```

```
int ReadNumber(int *value) {
    int ch, total;

    ch = ReadSymbol();

    if (ch >= '0' && ch <= '9') {
        total = ch - '0';
        *value = total;
        return TRUE;
    } else {
        UnReadSymbol(ch);
        return FALSE;
    }
}
```

## ReadNumber (continued)

```
int main(void) {
    int x;

    printf("Enter a number: ");
    if (!ReadNumber(&x)) {
        ParseError("Number Expected");
        return EXIT_FAILURE;
    }
    /* x now contains the number just read */
    printf("\nx = %d\n", x);
    return EXIT_SUCCESS;
}
```

## Overview: Uses of & and \*

```
int *p;
```

Definition of a pointer variable

```
p = &a;
```

Take the address of a and store in the pointer variable p

```
int b = *p;
```

*Dereference* p: Store in b the value of the variable that pointer variable p points to