CSE101-Lec#23 - 24

Pointers in C

Created By:
Amanpreet Kaur &
Sanjeev Kumar
SME (CSE) LPU

Outline

- Passing pointer to a Function
- Pointer and one dimensional Array

Introduction

- There are two ways to pass arguments to a function
  - **call-by-value and**
  - **call-by-reference.**
- All arguments in C are passed by value.
- Return may be used to return one value from a called function to a caller (or to return control from a called function without passing back a value).
- Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the over- head of passing the object by value.

Calling Functions by Reference

- Pointer arguments are used in call by reference.
- When calling a function with arguments that should be modified, the addresses of the arguments are passed.
  - Pass address of argument using & operator in function call.
  - Allows you to change actual location in memory
  - Arrays are not passed with & because the array name is already a pointer

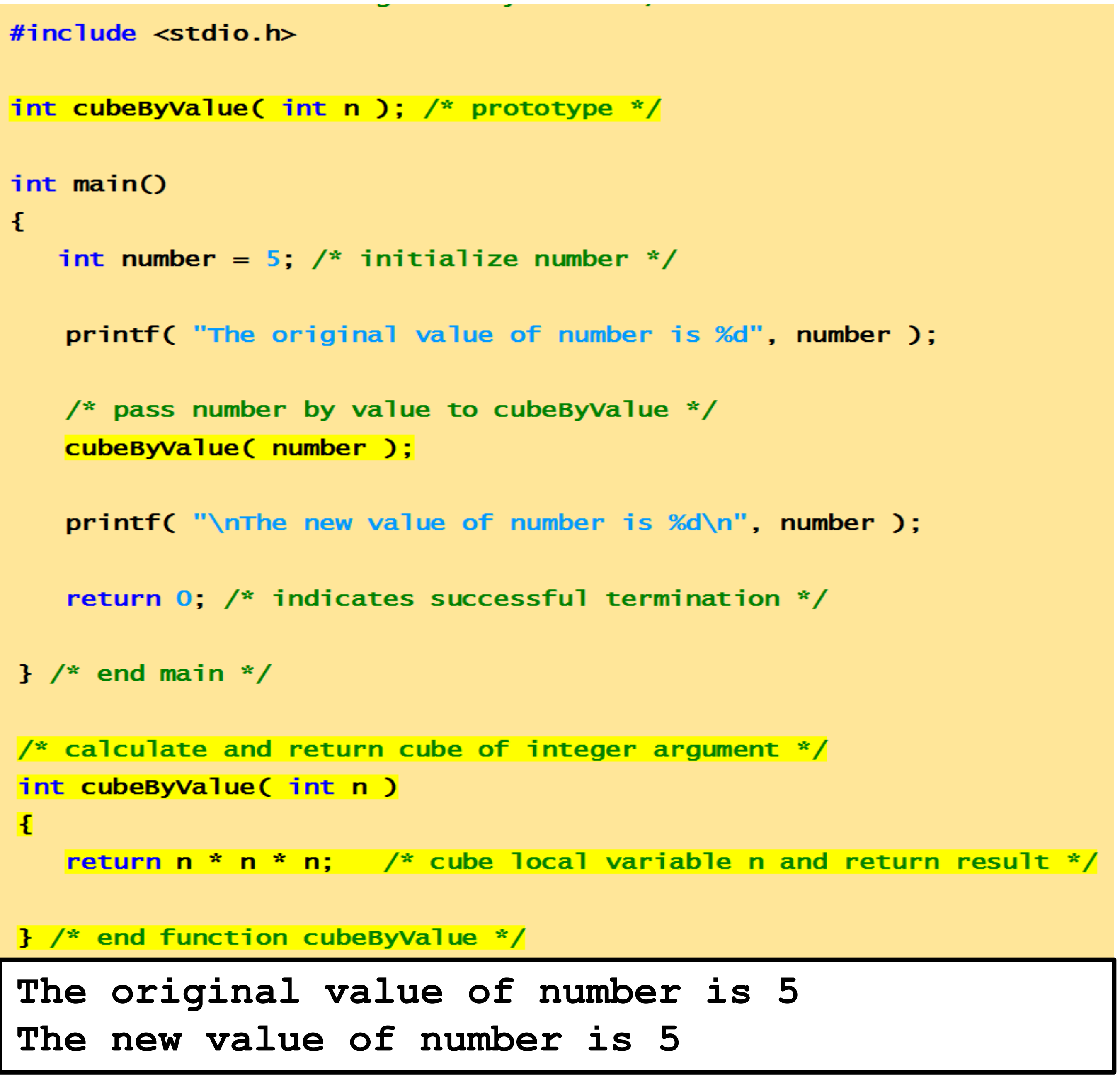
Calling Functions by Reference

- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.
- * operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number ) {
  *number = 2 * ( *number );
 }
```

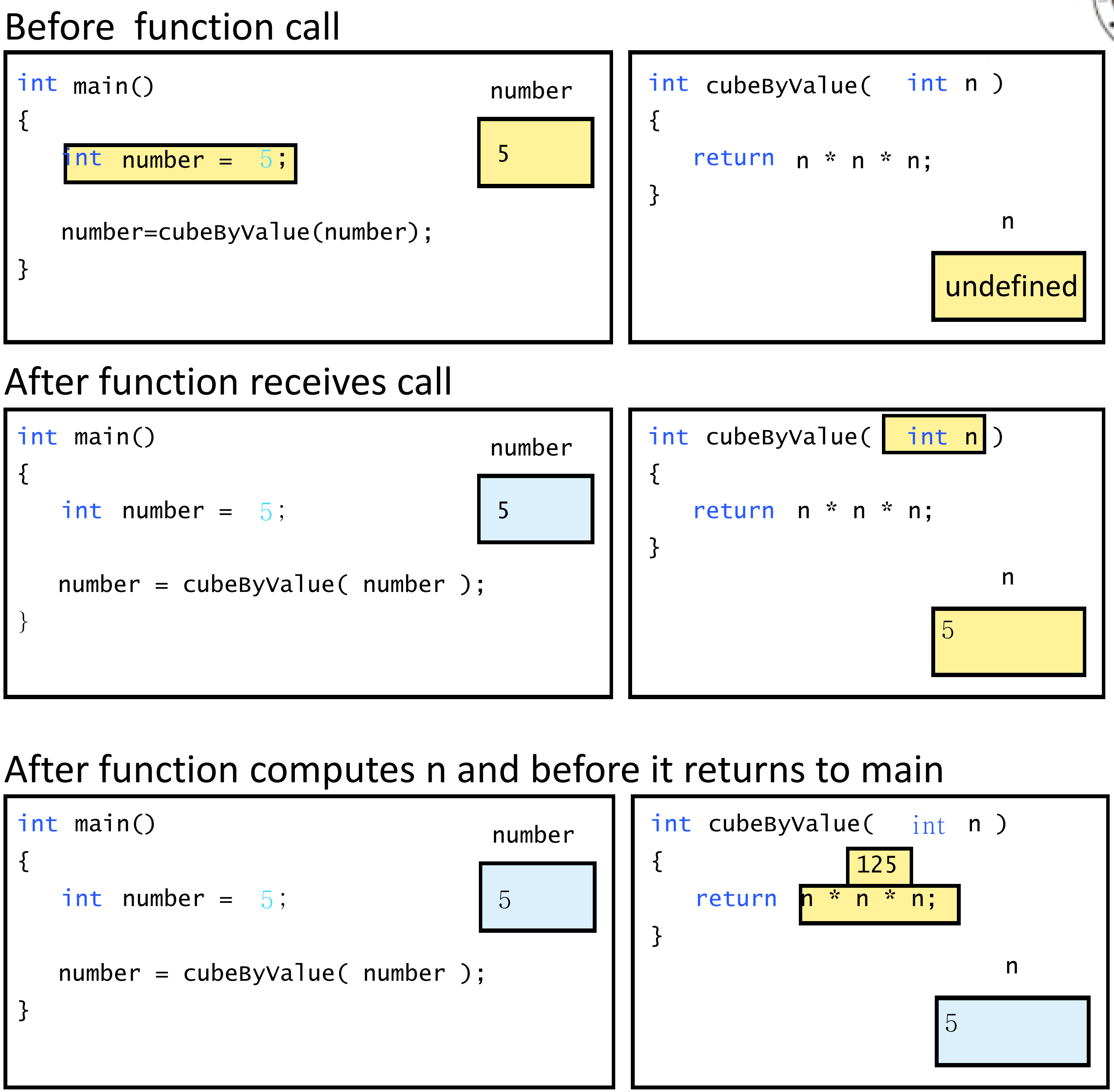
  - *number used as nickname for the variable passed

Program to call function by value



```
#include <stdio.h>

int cubeByValue( int n ); /* prototype */

int main()
{
   int number = 5; /* initialize number */

    printf( "The original value of number is %d", number );

    /* pass number by value to cubeByValue */
    cubeByValue( number );

    printf( "\nThe new value of number is %d\n", number );

    return 0; /* indicates successful termination */

 } /* end main */

 /* calculate and return cube of integer argument */
 int cubeByValue( int n )
 {
    return n * n * n;    /* cube local variable n and return result */

 } /* end function cubeByValue */
```

```
The original value of number is 5
The new value of number is 5
```



Before function call

```
int main()
{
   int number =  5;

   number=cubeByValue(number);
}
```

number

5

```
int cubeByValue(  int n )
{
   return n * n * n;
}
```

n

undefined

After function receives call

```
int main()
{
   int number =  5;

   number = cubeByValue( number );
}
```

number

5

```
int cubeByValue(  int n )
{
   return n * n * n;
}
```

n

5

After function computes n and before it returns to main

```
int main()
{
   int number =  5;

   number = cubeByValue( number );
}
```

number

5

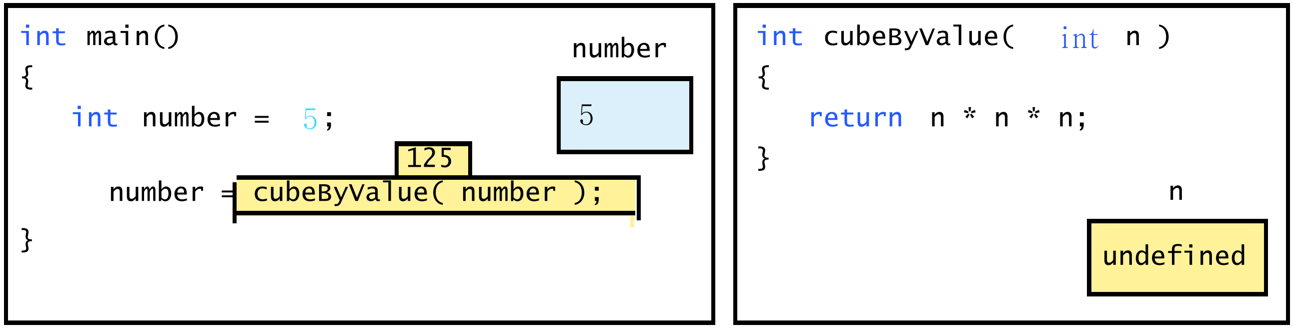
```
int cubeByValue(  int n )
{
            125
   return n * n * n;
}
```

n

5

Fig. Analysis of a typical call-by-value. (Part 1 of 2.)

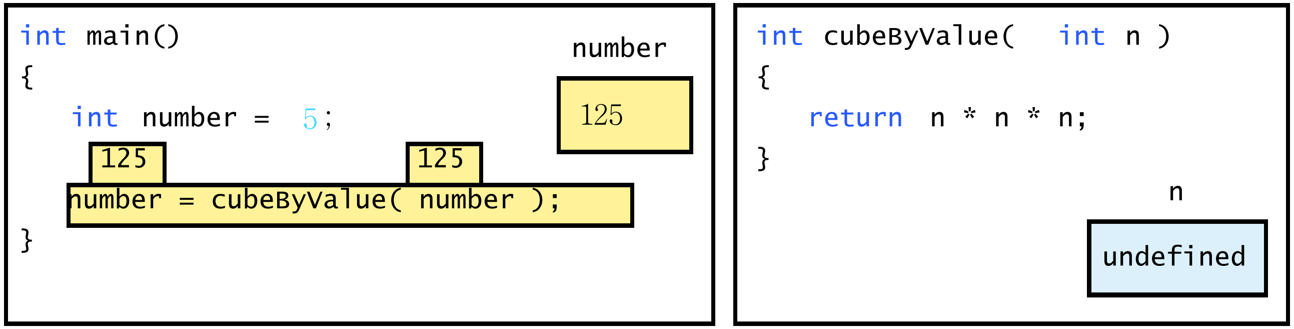
After function returns to main and before assigning result to number



```
int main()
{
   int number = 5;

      number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

After main completes the assignment to number

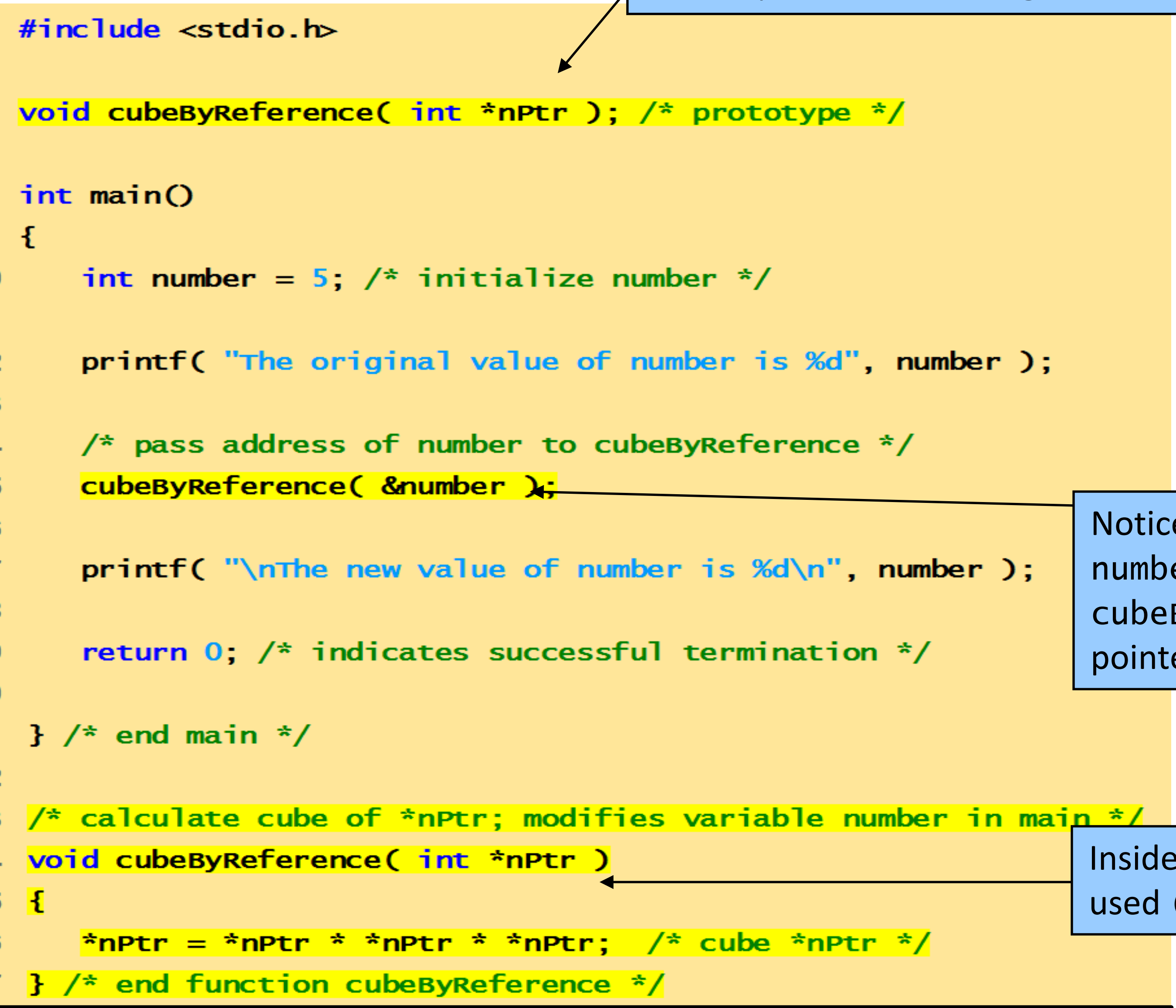


```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

Program to call a function by reference.

Notice that the function prototype takes a pointer to an integer.

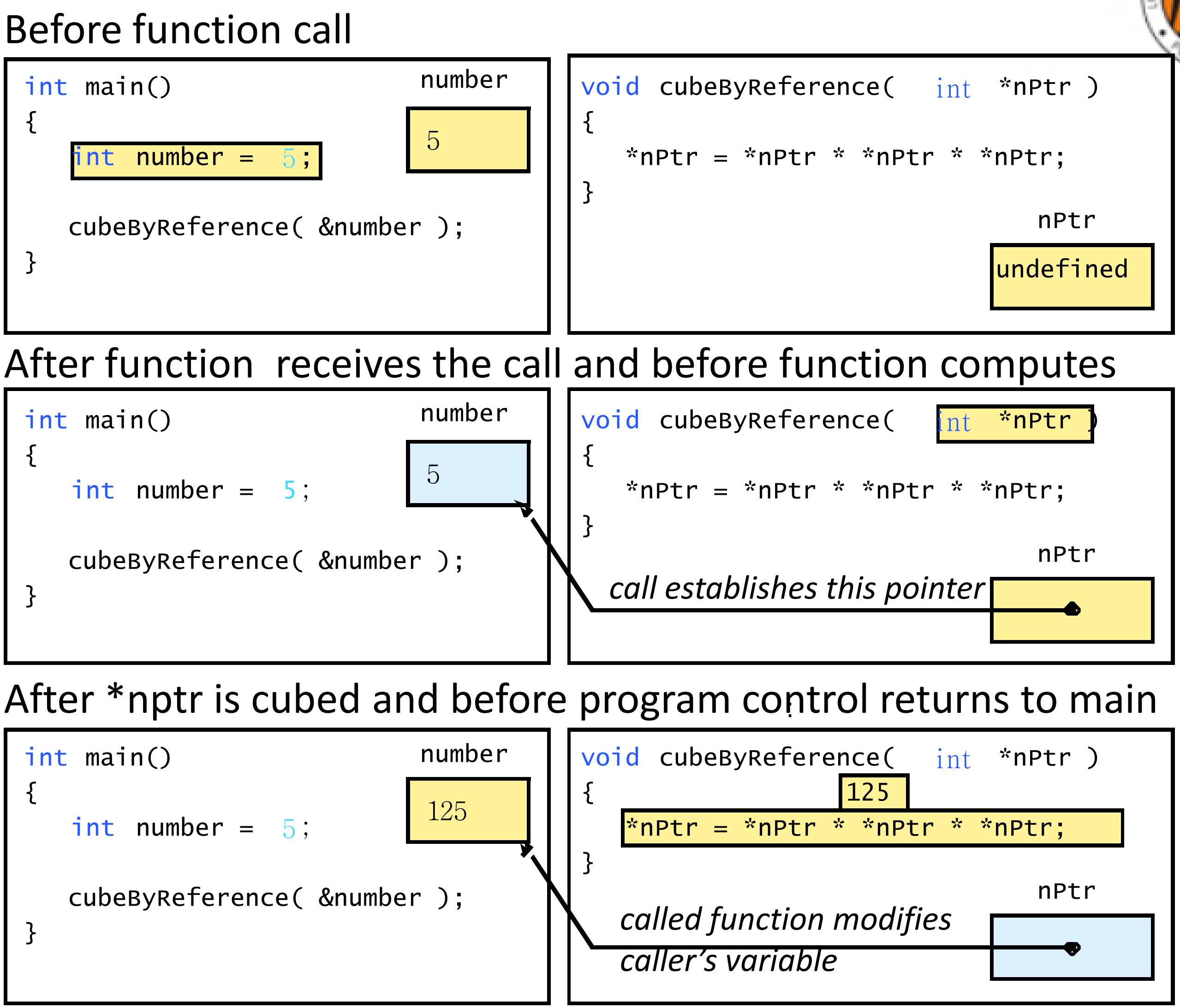


```
#include <stdio.h>

void cubeByReference( int *nPtr ); /* prototype */

int main()
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass address of number to cubeByReference */
   cubeByReference( &number );

   printf( "\nThe new value of number is %d\n", number );

   return 0; /* indicates successful termination */

} /* end main */

/* calculate cube of *nPtr; modifies variable number in main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
} /* end function cubeByReference */
```

Notice how the address of number is given - cubeByReference expects a pointer (an address of a variable).

Inside cubeByReference, *nPtr is used (*nPtr is number).

```
The original value of number is 5
The new value of number is 125
```



Before function call

```
int main()                                number
{                                         5
    int number =  5;

    cubeByReference( &number );
}
```

```
void cubeByReference(  int  *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
                                     nPtr
                                     undefined
```

After function receives the call and before function computes

```
int main()                                number
{                                         5
    int number =  5;

    cubeByReference( &number );
}
```

```
void cubeByReference(  int  *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
                                     nPtr
```

call establishes this pointer

After *nptr is cubed and before program control returns to main

```
int main()                                number
{                                         125
    int number =  5;

    cubeByReference( &number );
}
```

```
void cubeByReference(  int  *nPtr )
{                   125
   *nPtr = *nPtr * *nPtr * *nPtr;
}
                                     nPtr
```

called function modifies caller’s variable

Fig. Analysis of a typical call-by-reference with a pointer argument.

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name is like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[5]` and a pointer `bPtr`
  - To set `bPtr` to point to `b[5]`:

    ```
    bPtr = b;
    ```

    - The array name (b) is actually the address of first element of the array `b[5]` which is equivalent to

      ```
      bPtr = &b[0]
      ```

    - Explicitly assigns `bPtr` to address of first element of b

The Relationship Between Pointers and Arrays

- Element `b[3]`
  - Can be accessed by `*(bPtr + 3)`
    - Where `3` is the offset. Called **pointer/offset notation**
  - Can be accessed by `bptr[3]`
    - Called **pointer/subscript notation**
    - `bPtr[3]` same as `b[3]`
  - Can be accessed by performing pointer arithmetic on the array itself

    `*(b + 3)`

```c
#include <stdio.h>

int main()
{
   int b[] = { 10, 20, 30, 40 }; /* initialize array b */
   int *bPtr = b;                /* set bPtr to point to array b */
    /* output array b using array subscript notation */
    printf( "Array b printed with:\nArray subscript notation\n" );
       printf( "b[ %d ] = %d\n", 0, b[0] );

    /* output array b using array name and pointer/offset notation */
    printf( "\nPointer/offset notation where\n"
            "the pointer is the array name\n" );
    printf( "*( b + %d ) = %d\n", 1, *(b + 1) );

   /* output array b using bPtr and array subscript notation */
    printf( "\nPointer subscript notation\n" );
      printf( "bPtr[ %d ] = %d\n", 2, bPtr[2] );

   /* output array b using bPtr and pointer/offset notation */
    printf( "\nPointer/offset notation\n" );
      printf( "*( bPtr + %d ) = %d\n", 3, *(bPtr + 3) );
} /* end main */
```

Program to show subscripting and pointer notation with arrays.

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10

Pointer/offset notation where
the pointer is the array name
*( b + 1 ) = 20

Pointer subscript notation
bPtr[ 2 ] = 30

Pointer/offset notation
*( bPtr + 3 ) = 40
```

Character arrays and pointers

- What’s the difference between char s[] and char* s?

char s[] = “hello”

- Allocates the string in modifiable memory, and defines s to be a pointer to the head of the string.
- Can change the contents, but s will always point to the same place
- Can’t write: s = p; an array name is not a variable (i.e., can’t be used as an l-value)

char* s = “hello”

- Allocates a pointer (freely modifiable)
- Allocates a string (not modifiable)
- s points to the beginning of the string, but modifications to the string (e.g., *s = ‘x’) is undefined
- s can be reassigned to point to other strings

Next Class: Array of pointers

cse101@lpu.co.in