Lecture 8: C Pointers

Content

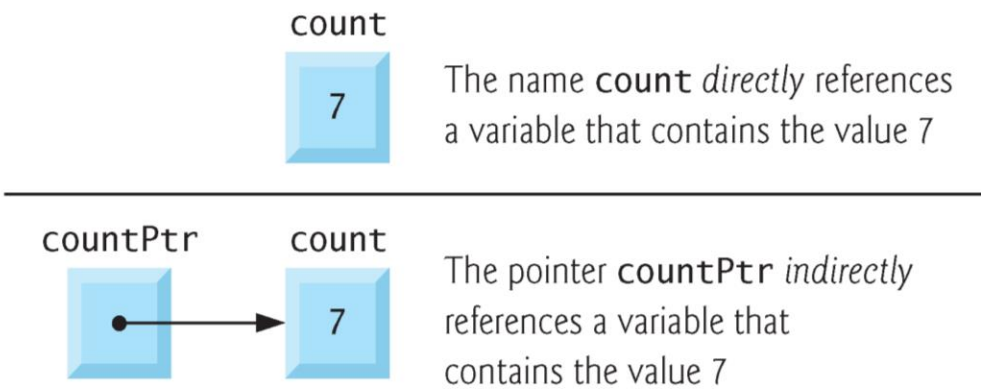
I. Pointer Definition
II. Pointer Variable Definition
III. Pointer in Expressions
IV. Passing Arguments to Functions by Reference
V. Pointers and Arrays
VI. Pointers and Structures

Reference: Deitel, C How to Program (8th ed.)

Dr Obada Al-Khatib slides

I. Pointer Definition

- **A Pointer** is a **variable whose value** is a ***memory address***.
  - Normally, a variable directly contains a specific value.
  - A **pointer,** on the other hand, **contains an *address* of a variable that contains a specific value.**
- In this sense, a **variable name *directly* references a value**, and a **pointer *indirectly* references a value**
- Referencing a value through a pointer is called **indirection or dereference**



Why it is Useful?

- **Pointer** is one of the most powerful features of the C programming language.
- Pointers enable programs to simulate **pass-by-reference**, to **pass functions between functions**, and to **create and manipulate dynamic data structures**, i.e., data structures that can **grow** and **shrink** at **execution time**, such as linked lists, queues, stacks and trees.
- In this subject, we focus only on **pass-by-reference.**

II. Pointer Variable Definition

- Pointers, like all variables, must be defined before they can be used.
- Each pointer has a type associated with it
- The definition
  - `int *countPtr, count;`

  specifies that,
  - variable countPtr is of type **int *** (i.e., a pointer to an integer) and is read (right to left), "countPtr is a pointer to int" or "countPtr points to an object of type int."
  - the variable count is defined to be an int, *not* a pointer to an int.

Pointer Variable Initialization

- **Before being used, pointer should be assigned an address:**
  - Pointers should be initialized when they're defined or they can be assigned an address value afterwards.
  - A pointer may be initialized to `NULL` or an address.
    - A pointer with the value `NULL` points to *nothing*.

Pointer Operators : (&, *)

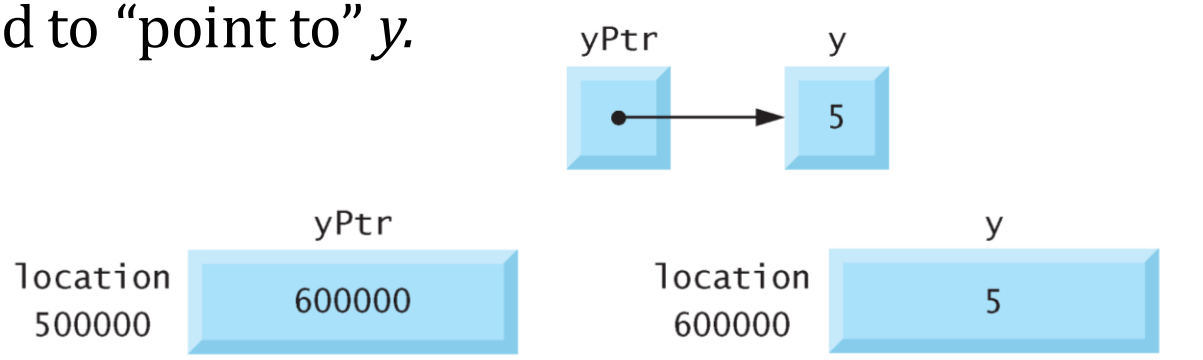
- The &, address operator, is a **unary** operator that returns the address of its operand.
- For example, assuming the definitions:
  - ```
    int y = 5;
    int *yPtr;
    ```

  the statement

  ```
  yPtr = &y;
  ```

  assigns the ***address*** of the variable *y* to pointer variable *yPtr*.

  Variable *yPtr* is then said to "point to" *y*.



Representation of y and yPtr in memory.
  ```

...Continued

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, **returns the *value*** of the object to the pointer points to.

- For example, the statement
  - `printf("%d", *yPtr);`

  prints the value of variable y, see last slide, namely 5.

- Using * in this manner is called **dereferencing a pointer**.

Example

```
2  // Using the & and * pointer operators.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     int a = 7;
8     int *aPtr = &a; // set aPtr to the address of a
9
10    printf("The address of a is %p"
11           "\nThe value of aPtr is %p", &a, aPtr);
12
13    printf("\n\nThe value of a is %d"
14           "\nThe value of *aPtr is %d", a, *aPtr);
15
16    printf("\n\nShowing that * and & are complements of "
17           "each other\n&*aPtr = %p"
18           "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
19 }
```

specifier %p outputs the memory location as a *hexadecimal* integer on most platforms

the *address* of a and the *value* of aPtr are identical

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

The & and * operators are complements of one another—when they're both applied consecutively to aPtr in either order, the same result is printed.

One Pointer with many Variables

- The same pointer can point to different data variables (one at a time) of the same data type:

Example:

```
float oldPrice =45.50, newPrice=30.99;
float *ptrP;
ptrP=&oldPrice;  /*obtain the address of oldPrice */
printf("The price %f\n", *ptrP);
ptrP=&newPrice; /*obtain the address of oldPrice */
prinftf("The price %f\n", *ptrP);
```

Display a value of type float pointed by **ptrP**

Output:

```
The price = 45.50
The price = 30.99
```

III. Pointers in Expressions

- Pointers are **valid operands in arithmetic expressions, assignment expressions and comparison expressions**.
- A pointer can be **assigned** to another pointer **if both have the same type.**
  - **Otherwise, you get warning which can be resolved using casting**
- The exception to the above rule is the pointer to void (i.e., `void *`), which is a **generic pointer** that can represent *any* pointer type.
  - All pointer types can be assigned a pointer to **`void`**, and a pointer to `void` can be assigned a pointer of any type.
    - In both cases, a **cast** operation is **not** required.
  - A pointer to `void` *cannot* be **dereferenced.**

Arithmetic Expression

- **A limited set of arithmetic operations may be performed on pointers.**
  - A pointer may be *incremented* (++) or *decremented* (--), an integer may be *added* to a pointer (+ or +=), an integer may be *subtracted* from a pointer (- or -=) and one pointer may be subtracted from another
    - Substraction is meaningful only when *both* pointers point to elements of the *same* array.

Addition with Pointers

- **Addition**: If one of the operand is a pointer the other one **must be an integer**

```
ptr2 = ptr1++;              /* CORRECT */
ptr2 = ptr1+100;            /* CORRECT */
int Add = ptr1+ptr2;        /* WRONG */
ptr3 = ptr1 + ptr2;         /* WRONG */
```

- The meaning of **addition with pointers** is **different** from **normal arithmetic addition** as **the second constant addend** is **scaled by the pointer's datatype number of bytes.**

type *ptr;

ptr++ or ptr+=1 is actually equivalent to ptr=ptr + **sizeof**(type)

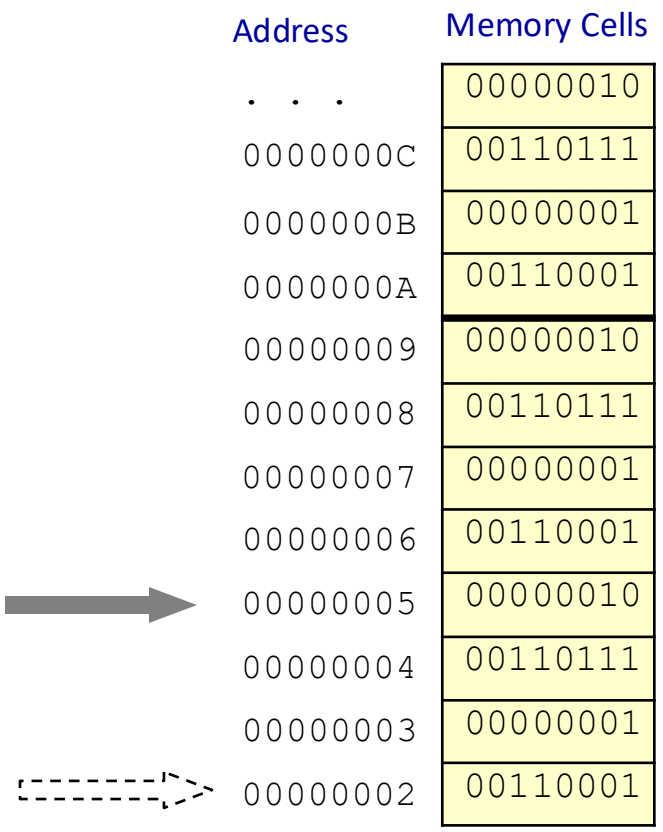
Example: Pointers to char

- Since `sizeof(char)` is `1`, the scale is 1

Example:

```
char *ptr;
char chVar;
ptr = &chVar;

ptr++;
ptr += 3;
ptr--;
```

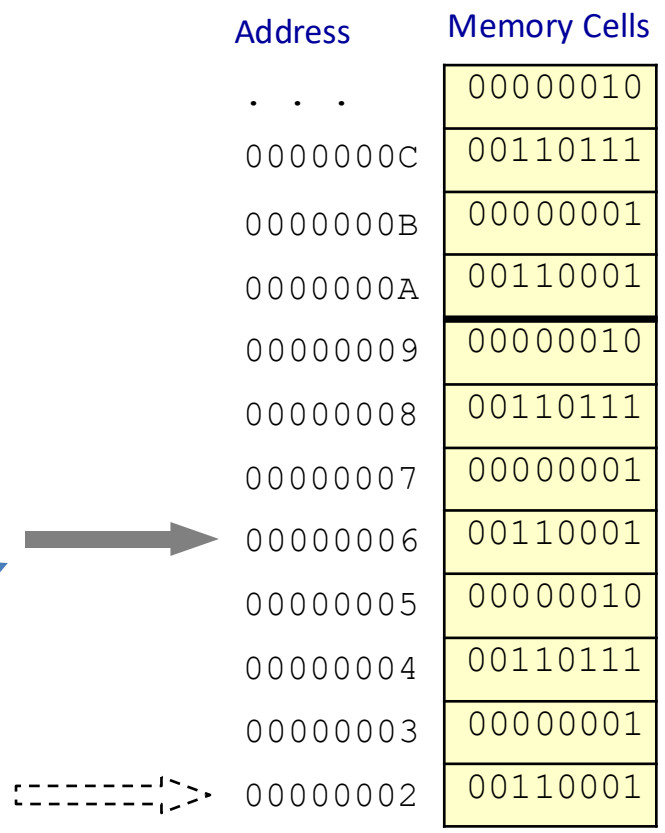


Example: Pointers to int

- Since `sizeof(int)` is **4**, the scale is 4

Example:

```
int *ptr;
int inVar;
ptr = &inVar;
ptr++;
ptr += 1;
ptr--;
```



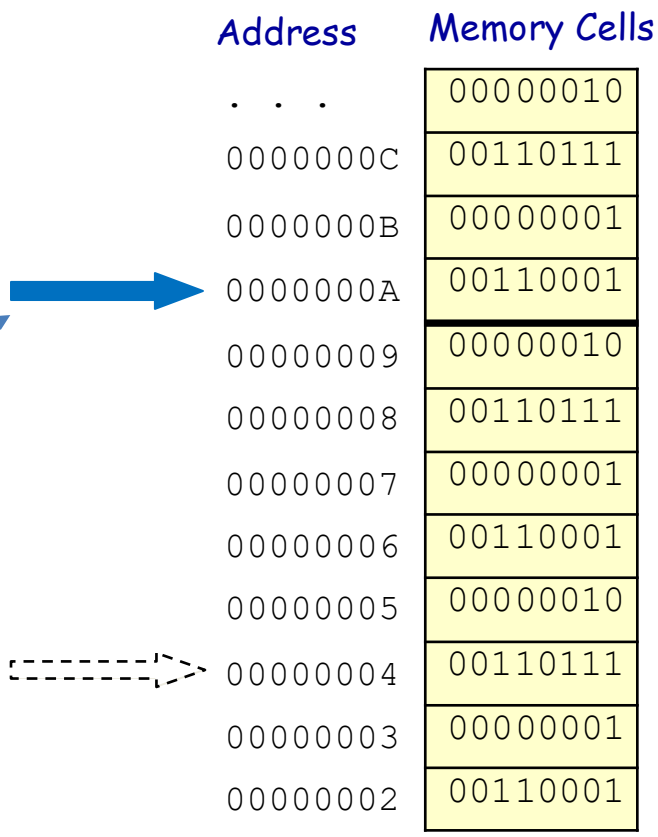
UNIVERSITY OF WOLLONGONG IN DUBAI

Pointer to short (2 bytes)

- Since `sizeof(short)` is `2`, the scale is 2

```
short int *ptr;
short int inVar;
ptr = &inVar;

ptr += 3;
```



Subtraction

```
type *ptr;
```

- Decrement

  is actually equivalent to

`ptr--;` or `ptr -= 1;`

`ptr - sizeof(type)`

- If both operands are pointers, the result is **integer**

```
int diff = ptr1-ptr2; /* OK */
ptr3 = ptr1 - ptr2; /* WRONG */
```

Example

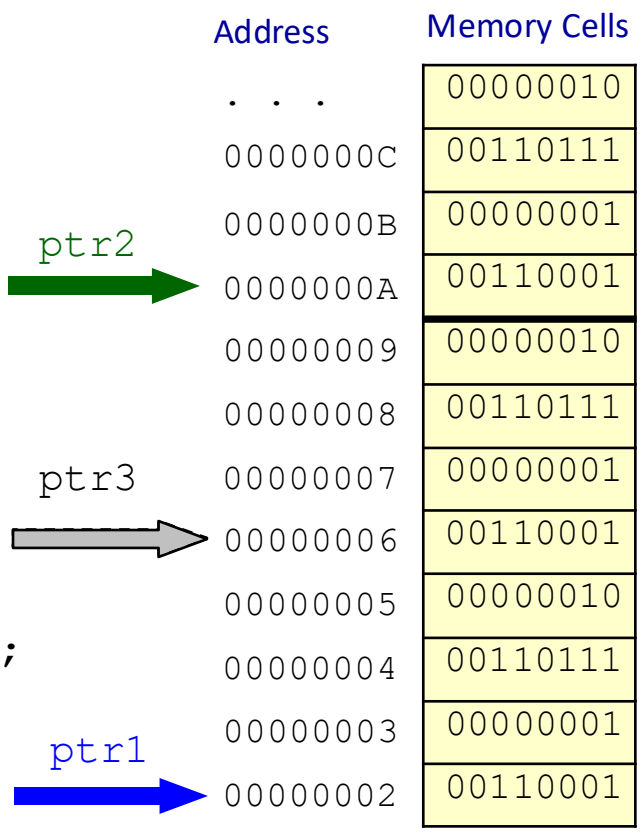
- How to point ptr3 right in the middle between ptr1 and ptr2?

```
char *ptr1,*ptr2, *ptr3;
```

Can you do this?

~~ptr3 = (ptr2 + ptr1)/2;~~

```
ptr3 = ptr1 + (ptr2 - ptr1)/2;
```



Multiplication/Division with Pointers

- Multiplication and division are NOT defined for pointers

```
type *ptr;

ptr = ptr * 3;          /* WRONG */

ptr = ptr / 2;          /* WRONG */

ptr = ptr % 2;          /* WRONG */
```

Relational Operations with Pointers

- You can use relational operators (<, ==, >, !=) with pointers providing that both operands **are pointers of the same type.**
  - **Otherwise, you get warning which can be resolved using casting**
- Example:

```
char *ptr1, *ptr2;
int *ptr3, *ptr4;
```

```
if( ptr1 <=    ptr2)    {  .  .  .  }        OK
if( ptr3 !=    NULL ) {    .  .  .  }        OK
while( ptr3    < ptr4){    .  .  .  }        OK

if( ptr1 ==    ptr3 ) {    .  .  .  }        Warning
                                             different types

 while( ptr2 < 0x2400FA ) { . . . }          Warning
```

Cannot compare a pointer with a fixed value / constant

Pointers Casting

- Simple data types

```
char  numCh = 25;
int   numIn;
float numFl;
numIn = (int)numCh + 7;
numFl = (float)numIn/2.0;
```

- Pointers

```
char  *ptrCh;
int   *ptrIn;

ptrIn = (int*) ptrCh;     /*  convert char* into int*  */
ptrCh = (char*) ptrIn;    /*  convert int* into char*  */
```

It is a valid operation, but make sure you understand what you're doing and how this may affect your program before casting pointers

Quiz1: Pointer Arithmetic

- What value is printed out ?

```
short int number = 5, result;
short int *ptr = NULL;

ptr = &number;
printf("%d", number);

*ptr = 2;
printf("%d", number);

result = 10+*ptr;
printf("%d", result);

result = 10/*ptr;
printf("%d", result);
```

5

2

12

5

...Continued

- How are these statements interpreted ?

```
*ptr++ = 10;

(*ptr)++;

*(++ptr);

++(*ptr);
```

- See appendix for operators precedence
- *ptr++ = 10; we assign 10 to the variable that is pointed to by ptr and then increase the pointer by one.
- (*ptr)++; increase the value of the variable appointed to by ptr by one.
- *(++ptr); This increases the physical address by 2 (short `int *ptr = NULL;`) and then points to that address. Doesn't change the value of the variable appointed to by ptr.
- ++(*ptr); increase the value of the variable appointed to by ptr by one.

Quiz 2

- What is wrong with this code?

```
float voltage = 3.3;
float *ptrV;
ptrV =  voltage;
```

ptrV = &voltage

- What is the value printed?

```
int number = 10;
int *ptr;

ptr = &number;
*ptr = *ptr + 2;
printf("%d", number);
```

12

- What is wrong with this code?

```
unsigned char timeSec;
char  *ptr1;
```

Fine, data type of the same size

```
ptr1 = &timeSec;
```

```
int *ptr1,
float *ptr2;

ptr1 = ptr2;
```

Warning:
two pointers of different type

IV. Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—**pass-by-value** and **pass-by-reference.**
- **All arguments in C** *are* ***passed by value.***
- Many functions require the capability to ***modify variables in the caller*** or to **pass a pointer to a large data object** to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).

- In C, **use pointers and the indirection operator(*)** to **simulate pass-by-reference.**

...Continued

- **C Passing by reference:** when **calling a function** with **arguments that should be modified**, the ***addresses*** **of the arguments are passed.**
  - This is normally accomplished by applying the **address operator (&)** to the variable (in the caller) whose value will be modified.
  - **Arrays are *not* passed using operator &** because C automatically passes the starting location in memory of the array (**the name of an array is equivalent to `&arrayName[0]`**).
  - 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.**
  - A function **receiving an *address* as an argument must define a *pointer parameter* to receive the address.**

Example: Pass-By-Value (1)

```
2  // Cube a variable using pass-by-value.
3  #include <stdio.h>
4
5  int cubeByValue(int n); // prototype
6
7  int main(void)
8  {
9     int number = 5; // initialize number
10
11    printf("The original value of number is %d", number);
12
13    // pass number by value to cubeByValue
14    cubeByValue(number);
15
16    printf("\nThe new value of number is %d\n", number);
17 }
18
19 // calculate and return cube of integer argument
20 int cubeByValue(int n)
21 {
22    return n * n * n; // cube local variable n and return result
23 }
```

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

Example: Pass-By-Value (2)

```
2  // Cube a variable using pass-by-value.
3  #include <stdio.h>
4
5  int cubeByValue(int n); // prototype
6
7  int main(void)
8  {
9     int number = 5; // initialize number
10
11    printf("The original value of number is %d", number);
12
13    // pass number by value to cubeByValue
14    number = cubeByValue(number);
15
16    printf("\nThe new value of number is %d\n", number);
17 }
18
19 // calculate and return cube of integer argument
20 int cubeByValue(int n)
21 {
22    return n * n * n; // cube local variable n and return result
23 }
```

The new value is assigned to `number` in `main`

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

Example: Pass-By-Reference

```
2  // Cube a variable using pass-by-reference with a pointer argument.
3
4  #include <stdio.h>
5
6  void cubeByReference(int *nPtr); // function prototype
7
8  int main(void)
9  {
10    int number = 5; // initialize number
11
12    printf("The original value of number is %d", number);
13
14    // pass address of number to cubeByReference
15    cubeByReference(&number);
16
17    printf("\nThe new value of number is %d\n", number);
18 }
19
20 // calculate cube of *nPtr; actually modifies number in main
21 void cubeByReference(int *nPtr)
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 }
```

reference—the address of number is passed

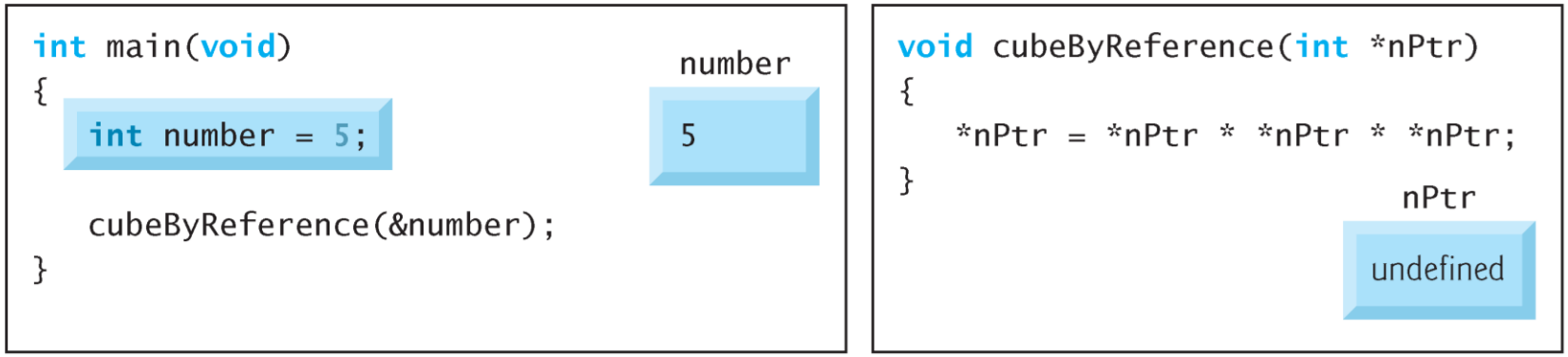
Function does not return a value.

a pointer to an int

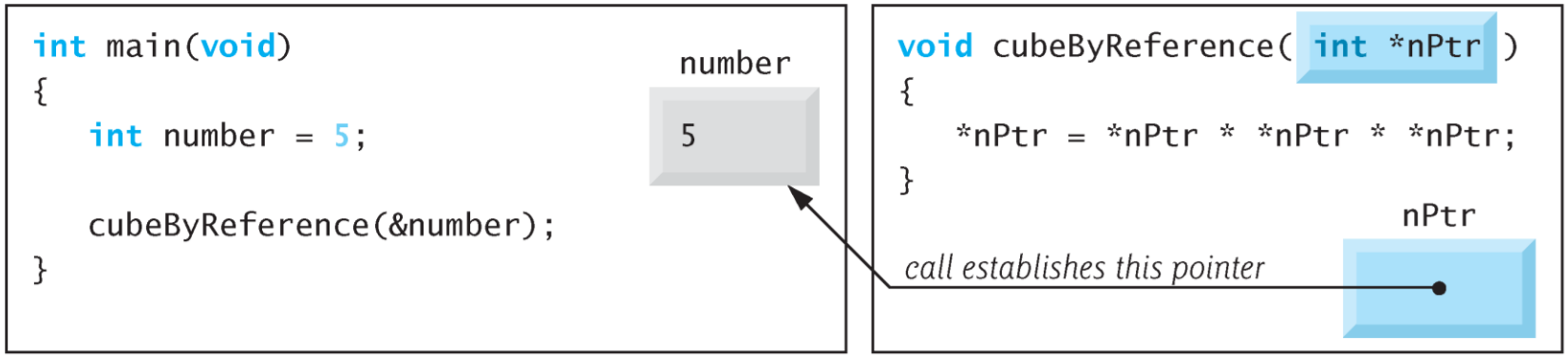
It is the number in main

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

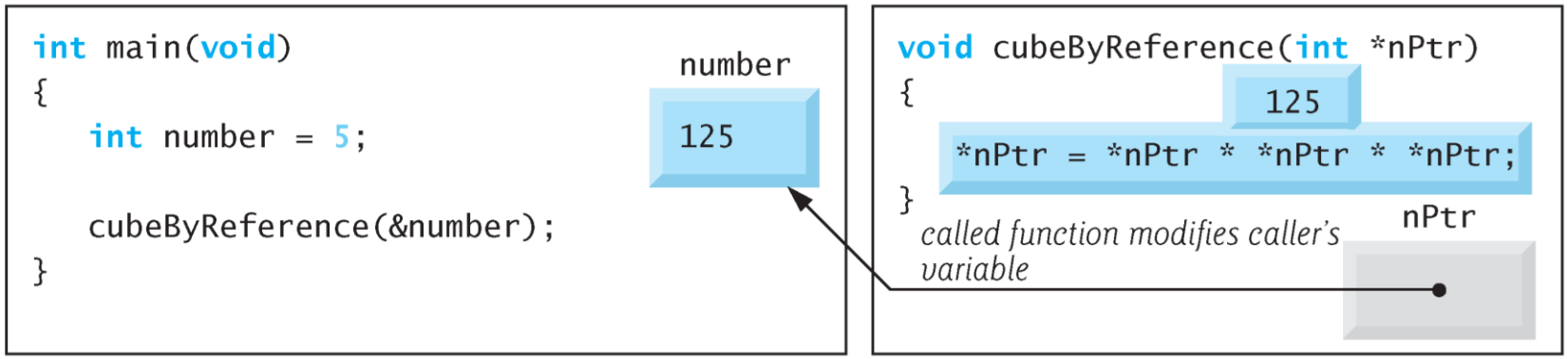
Step 1: Before `main` calls `cubeByReference`:



Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



Step 3: After `*nPtr` is cubed and before program control returns to `main`:



...Continued

- A function can take pointers as parameters and also can return a value of a pointer type which is one of the function parameters list
- Examples:

  void **order (**double ***smp,** double ***lgp)**;

  int **fwrite**(void ***buffer**,int size, int num);

- Although a function cannot <u>return</u> more than one value, with the use of pointers you can <u>pass back</u> as many values as you need

Example:

```
bool update( float *price, int *quantity, char *code );
```

These three function parameters can be updated and passed back to the top level function

Example : Function with Output Parameters

- Write a function `separate`, which finds the sign (`signp`), whole number magnitude (`wholep`), and fractional parts (`fracp`) of its first parameter.
  - In function `separate`, only the first formal parameter, `num` , is an input; the other three formal parameters (`signp, wholep,` and `fracp)` are output parameters, used **to carry multiple results from** function `separate` back to the function calling it.

...Continued

```c
/*
 * Demonstrates the use of a function with input and output parameters.
 */

#include <stdio.h>
#include <math.h>
void separate(double num, char *signp, int *wholep, double *fracp);

int
main(void)
{
      double value; /* input - number to analyze                          */
      char sn;      /* output - sign of value                             */
      int whl;      /* output - whole number magnitude of value           */
      double fr;    /* output - fractional part of value                  */

      /* Gets data                                                         */
      printf("Enter a value to analyze> ");
      scanf("%lf", &value);

      /* Separates data value into three parts                             */
      separate(value, &sn, &whl, &fr);

      /* Prints results                                                    */
      printf("Parts of %.4f\n sign: %c\n", value, sn);
      printf(" whole number magnitude: %d\n", whl);
      printf(" fractional part: %.4f\n", fr);

      return (0);
}
```

Pointers

Pass by reference

...Continued

```
/*
 * Separates a number into three parts: a sign (+, -, or blank),
 * a whole number magnitude, and a fractional part.
 * Pre: num is defined; signp, wholep, and fracp contain addresses of memory
 *      cells where results are to be stored
 * Post: function results are stored in cells pointed to by signp, wholep, and
 *       fracp
 */
void
separate(double  num,      /* input - value to be split                    */
         char   *signp,    /* output - sign of num                         */
         int    *wholep,   /* output - whole number magnitude of num       */
         double *fracp)    /* output - fractional part of num              */
{
      double magnitude;   /* local variable - magnitude of num            */

      /* Determines sign of num */
      if (num < 0)
           *signp = '-';
      else if (num == 0)
           *signp = ' ';
      else
           *signp = '+';

      /* Finds magnitude of num (its absolute value) and separates it into
         whole and fractional parts                                        */
      magnitude = fabs(num);
      *wholep = floor(magnitude);
      *fracp = magnitude - *wholep;
}
```

```
Enter a value to analyze> 35.817
Parts of 35.8170
  sign: +
  whole number magnitude: 35
  fractional part: 0.8170
```

Quiz 3

- Show the table of values for x , y , and z that is the output displayed by the following program.

```
#include <stdio.h>
void sum(int a, int b, int *cp);
int
main(void)
{
     int x, y, z;
     x = 7; y = 2;
     printf(" x y z\n\n");

     sum(x, y, &z);
     printf("%4d%4d%4d\n", x, y, z);

     sum(y, x, &z);
     printf("%4d%4d%4d\n", x, y, z);

     sum(z, y, &x);
     printf("%4d%4d%4d\n", x, y, z);

     sum(z, z, &x);
     printf("%4d%4d%4d\n", x, y, z);

     sum(y, y, &y);
     printf("%4d%4d%4d\n", x, y, z);
     return (0);
     }
     void
     sum(int a, int b, int *cp)
     {
          *cp = a + b;
     }
```

```
 x   y   z

 7   2   9

 7   2   9

11   2   9

18   2   9

18   4   9
```

V. Pointers and Arrays

- **Arrays and pointers** are intimately related in C and **often may be used interchangeably.**
- An ***array name*** can be thought of as a **constant pointer.**
- **Pointers** can be used to **do any operation involving array indexing.**
  - Assume that integer array **b[5]** and integer **pointer variable bPtr** have been defined.
  - Because the array name (without an index) is a pointer to the first element of the array, we can set bPtr equal to the address of the first element in array b with the statement:

```
bPtr = b;
```

...Continued

- This statement **bPtr = b;** is equivalent to **bPtr = &b[0];**
- Array element b[3] can alternatively be referenced with the pointer expression *(bPtr + **3**)
  - the pointer variable points to the array's first element,
  - the offset 3 indicates which array element should be referenced,
  - the offset value is identical to the array index.
  - This notation is referred to as pointer/offset notation.

- The parentheses in the expression *(bPtr + **offset**) are **necessary because the precedence of * is higher than the precedence of +.**
  - Without the parentheses, the above expression would add offset to the value of the expression *bPtr (b[0]).

...Continued

- Just as the array element can be referenced with a pointer expression, the address &b[**offset**] can be written with the pointer expression: bPtr + **offset** **or b+offset**
  - **The array itself can be treated as a pointer and used in pointer arithmetic.**
  - The expression b+offset does not change the value of b, the array name, it is still pointing to the first element.

- Remember that an **array** name is essentially a **constant pointer**; it always points to the beginning of the array.
  - The expression: `b += 3` is ***invalid*** because it attempts to modify the value of the array name with pointer arithmetic.

Example

- To print the four elements of the integer array b, the following example uses the four methods we've discussed for referring to array elements:
  - array indexing,
  - offset with the array name as a pointer,
  - pointer indexing, and
  - offset with a pointer

typedef int size_t;

```
2  // Using indexing and pointer notations with arrays.
3  #include <stdio.h>
4  #define ARRAY_SIZE 4
5
6  int main(void)
7  {
8     int b[] = {10, 20, 30, 40}; // create and initialize array b
9     int *bPtr = b; // create bPtr and point it to array b
10
11    // output array b using array index notation
12    puts("Array b printed with:\nArray index notation");
13
14    // loop through array b
15    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
16       printf("b[%u] = %d\n", i, b[i]);
17    }
18
19    // output array b using array name and pointer/offset notation
20    puts("\nPointer/offset notation where\n"
21          "the pointer is the array name");
22
```

```
23      // loop through array b
24      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
25         printf("*(b + %u) = %d\n", offset, *(b + offset));
26      }
27
28      // output array b using bPtr and array index notation
29      puts("\nPointer index notation");
30
31      // loop through array b
32      for (size_t i = 0; i < ARRAY_SIZE; ++i) {
33         printf("bPtr[%u] = %d\n", i, bPtr[i]);
34      }
35
36      // output array b using bPtr and pointer/offset notation
37      puts("\nPointer/offset notation");
38
39      // loop through array b
40      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
41         printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
42      }
43   }
```

```
Array b printed with:
Array index notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer index notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings, referred to simply as a string array.
  - Each entry in the array is a string, but in C a string can be defined as a pointer to its first character.
  - So each entry in an array of strings is actually a pointer to the first character of a string.

...Continued

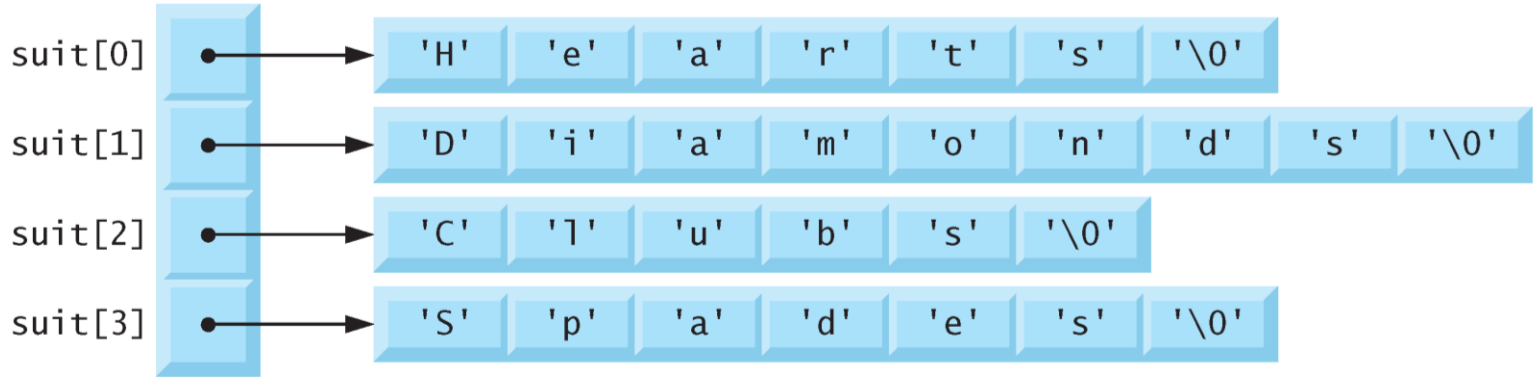
- Consider the definition of string array *suit*, which might be useful in representing a deck of cards.

```
const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

  - The suit[4] portion of the definition indicates an array of 4 elements.
  - The **char *** portion of the declaration indicates that each element of array *suit* is of type "pointer to char."
  - Qualifier **const** indicates that the strings pointed to by each element pointer will not be modified.
  - The four values to be placed in the array are "Hearts", "Diamonds", "Clubs" and "Spades".
  - Each is stored in memory as a ***null-terminated character string*** that's one character longer than the number of characters between quotes.

...Continued

- The four strings are 7, 9, 6 and 7 characters long, respectively.
- Although it appears as though these strings are being placed in the `suit` array, **only pointers are actually stored in the array**
- Each pointer points to the first character of its corresponding string.
- **Thus, even though the `suit` array is *fixed* in size, it provides access to character strings of *any length*.**



VI. Pointers and Structures

- You can declare pointers to structures:

```
typedef struct
{
int number;
int quantity;
} part_t;
```

```
part_t *ptrPart;
```

- Since part_t is a **structured data type definition** (not a declaration) you cannot use its pointer until a variable of the type part_t is declared

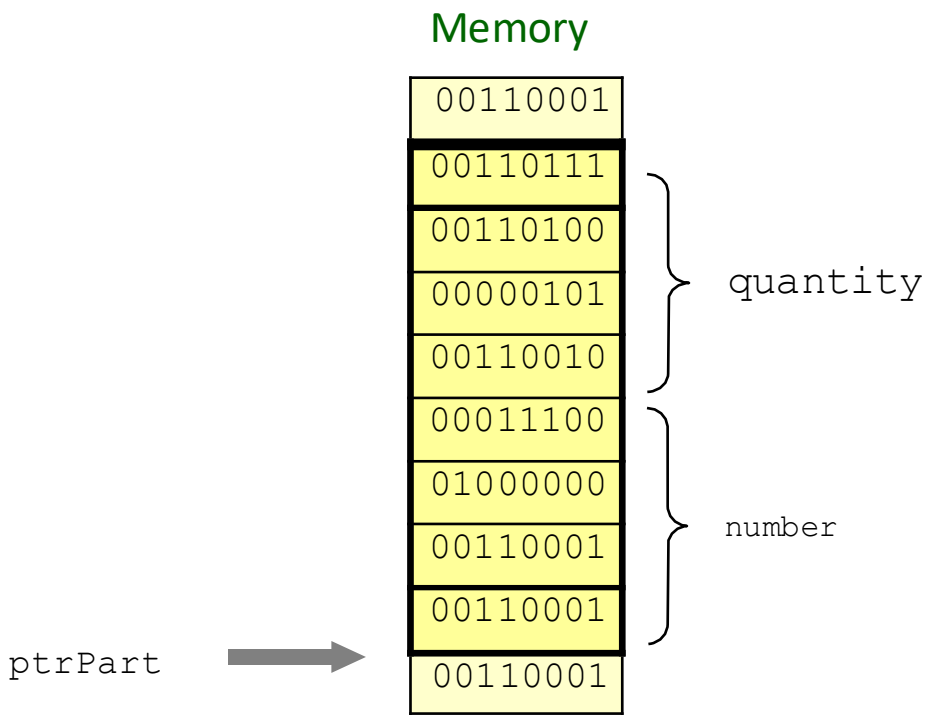
```
part_t bc209;
part_t * ptrPart = &bc209;    /* OK */
```

...Continued

- A pointer to a structured variable points to the base address of the structure

```
typedef struct
{
int number;
int quantity;
}part_6;
part_t *ptrPart;

part_t bc209;
ptrPart = &bc209;
```



Members Access

- Once a pointer points to a structure, you can access all members of the structure

```
typedef struct
{
   int number;
    int quantity;
}part_t;


part_t *ptrPart;
part bc209;


ptrPart = &bc209;
(*ptrPart).number =245603;
```

this is equivalent to

```
bc209.number = 245603;
```

...Continued

- Another way to access the structure members is to use **indirect membership operator ->**

```
typedef struct
{
   int number;
   int quantity;
}part_t;


part_t *ptrPart; part bc209;


ptrPart = &bc209;

ptrPart->number = 245603;
```

this is also equivalent to

```
bc209.number = 245603;
```

Struct Member Access - Summary

- Therefore, there are three ways of accessing members of a structured variable:

1.Using the variable name and the member access operator

```
bc209.number = 245603;
```

2.Using the indirection operator with a pointer and the member access operator

```
(*ptr).number = 245603;
```

3.Using a pointer followed by the indirect membership operator

```
ptr->number = 245603;
```

Example: Function with a Structured Input Parameters

```
1.  /*
2.   * Displays with labels all components of a planet_t structure
3.   */
4.  void
5.  print_planet(planet_t pl) /* input - one planet structure */
6.  {
7.        printf("%s\n", pl.name);
8.        printf("  Equatorial diameter: %.0f km\n", pl.diameter);
9.        printf("  Number of moons: %d\n", pl.moons);
10.       printf("  Time to complete one orbit of the sun: %.2f years\n",
11.              pl.orbit_time);
12.       printf("  Time to complete one rotation on axis: %.4f hours\n",
13.              pl.rotation_time);
14. }
```

```
#define STRSIZ 10

typedef struct {
      char   name[STRSIZ];
      double diameter;            /* equatorial diameter in km     */
      int    moons;               /* number of moons               */
      double orbit_time,          /* years to orbit sun once       */
             rotation_time;       /* hours to complete one
                                        revolution on axis         */
} planet_t;
```

...Continued

```
1.  /*
2.   * Fills a type planet_t structure with input data. Integer returned as
3.   * function result is success/failure/EOF indicator.
4.   *      1 => successful input of one planet
5.   *      0 => error encountered
6.   *      EOF => insufficient data before end of file
7.   * In case of error or EOF, value of type planet_t output argument is
8.   * undefined.
9.   */
10. int
11. scan_planet(planet_t *plnp) /* output - address of planet_t structure
12.                                to fill                              */
13. {
14.       int result;
15.
16.       result = scanf("%s%lf%d%lf%lf", (*plnp).name,
17.                                        &(*plnp).diameter,
18.                                        &(*plnp).moons,
19.                                        &(*plnp).orbit_time,
20.                                        &(*plnp).rotation_time);
21.       if (result == 5)
22.             result = 1;
23.       else if (result != EOF)
24.             result = 0;
25.
26.       return (result);
27. }
```

Another way by using the -> operator

```
result = scanf("%s%lf%d%lf%lf", plnp->name,
                                &plnp->diameter,
                                &plnp->moons,
                                &plnp->orbit_time,
                                &plnp->rotation_time);
```

```
#define STRSIZ 10

typedef struct {
      char   name[STRSIZ];
      double diameter;          /* equatorial diameter in km    */
      int    moons;             /* number of moons              */
      double orbit_time,        /* years to orbit sun once      */
             rotation_time;     /* hours to complete one
                                   revolution on axis           */
} planet_t;
```

Appendix: Operators Precedence

Operators	Associativity	Type
() [] ++ *(postfix)* -- *(postfix)*	left to right	postfix
+ - ++ -- ! * & (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > s>=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

https://en.cppreference.com/w/cpp/language/operator_precedence