

# 4 Pointers

1

## **Why Learning Pointers**

1. Pointer is a very powerful tool for the design of C programs. A pointer is a variable that holds the value of the address or memory location of another data object.
2. In C, pointers can be used in many ways. This includes the passing of variable's address to functions to support call by reference, and the use of pointers for the processing of arrays and strings.
3. In this lecture, we discuss the concepts of pointers including address operator, pointer variables and call by reference.

## Pointers

- **Primitive Data Types, Variables and Address Operator**
- Pointer Variables
- Call by Reference

2

### Pointers

1. We start by discussing primitive data types, variables and address operator.

## Variables of Primitive Data Types

```
#include <stdio.h>
int main()
{
    int num = 5;

    printf("num = %d, \n", num );
}
```

**Variables of primitive data types:**  
int, char, float, etc.

Printing the value of the variable

**Output**  
num = 5,

**Memory**

Address	Memory	Content of memory
1000	0 1 0 1 0 1 1 0	← 5
1001	1 0 1 0 0 1 0 1	
1002	0 1 0 1 1 0 1 0	
1003		
1004		
1005		
1006		
1007		
1008		
1009		

3

**Note:** The variable **num** stores the **value**.

### Variables of Primitive Data Types

1. A computer's memory is used to store data objects such as variables and arrays in C. Each memory location has an address that can hold one byte of information. They are organized sequentially and the addresses range from 0 to the maximum size of the memory.
2. When a variable is declared with a certain data type, the corresponding memory location will be allocated for the variable to hold the data of that type.
3. Note that variables of primitive data types such as **int**, **char**, **float**, **double**, etc. are used to store the actual data.
4. For example, in the program, the variable **num** is declared as an **int**. When the variable **num** is initialized with the value 5, the memory location of the variable is used to store the actual value of 5. When the variable **num** is printed with the **printf()** statement, the value 5 will then be printed on the screen.

## Variables of Primitive Data Types

```

#include <stdio.h>
int main()
{
    int num = 5;

    printf("num = %d, \n", num);
    scanf("%d", &num);
    printf("num = %d, \n", num);
}

```

**Printing the value of the variable**

→ **10**

**Output**  
num = 5,  
**10**  
num = 10,

**Note: The variable **num** stores the **value**.**

**Variables of primitive data types:**  
int, char, float, etc.

Address	Memory	num
1000	0 1 0 1 0 1 1 0	10
1001	1 0 1 0 1 0 1 0	
1002	0 1 0 1 0 1 0 0	
1003		
1004		
1005		
1006		
1007		
1008		
1009		
⋮	⋮	4

Content of memory

### Variables of Primitive Data Types

1. When the variable **num** is updated (for example, using **scanf()**) to the value of 10, the memory location of the variable is also updated to store the value of 10.
2. When the variable **num** is printed with the **printf()** statement, the value 10 will then be printed on the screen.

## Address Operator (&)

```
#include <stdio.h>
int main()
{
    int num = 5;

    printf("num = %d, &num = %p\n", num, &num);
    → scanf("%d", &num);
    printf("num = %d, &num = %p\n", num, &num);
}
```

Printing the  
memory  
address of the  
variable

**Output**

num = 5, &num = 1000 [address]

→ 10

num = 10, &num = 1000

Address	Memory	
1000	0 1 0 1 0 1 1 0	← Content of memory
1001	1 0 1 0 1 0 1 1	
1002	0 1 1 1 0 1 0 0	
1003		
1004		
1005		
1006		
1007		
1008		
1009		
⋮	⋮	

5

Note: The variable **num** stores the **value**.

### Address Operator

1. The address of a variable can be obtained by the *address operator (&)*. In the program, we can print the address of the variable **num** (i.e. **&num**). To do this, we need to use **%p** in the conversion specifier in the control string of the **printf()** statement: **printf("num = %d, &num = %p\n", num, &num);**
2. In the **printf()** statement, it prints two values on the screen. The first one is the value 5 that is the initialized value stored at the memory location of the variable **num**. The other is the memory address at which the value 5 is stored. The address of this memory location is 1000. However, as the memory location is assigned by the computer, it may be different every time the same program is run. We use **&num** to find the value of the address.
3. After executing the **scanf()** statement **scanf("%d", &num);** which reads in a value of 10 from the standard input, the value is then stored into the address location of the variable **num**.
4. When we perform the **printf()** statement again, the value stored in **num** has been updated to 10 due to user input. However, the memory address of the variable **num** remains the same throughout the execution of the program.

## Primitive Variables: Key Ideas

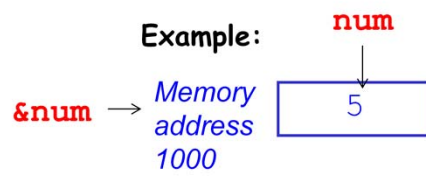
```
int num;
```

### (1) num

- it is a variable of data type int
- its memory location (4 bytes) stores the int value of the variable

### (2) &num

- it refers to the memory address of the variable
- the memory location is used to store the int value of the variable



**Note:** You may also print the address of the variable using the `printf()` statement.

6

### Primitive Variables: Key Ideas

1. There are two important ideas related to primitive variables in the above example:
  - a) The primitive variable (**num**) stores a variable value of data type int.
  - b) After applying the address operator to the variable **num** (i.e., **&num**), it refers to the memory address of the variable. The memory location is used to store the variable value.

## Pointers

- Primitive Data Types, Variables and Address Operator
- **Pointer Variables**
- Call by Reference

7

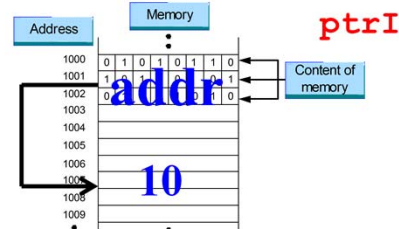
### Pointers

1. Here, we discuss pointer variables.

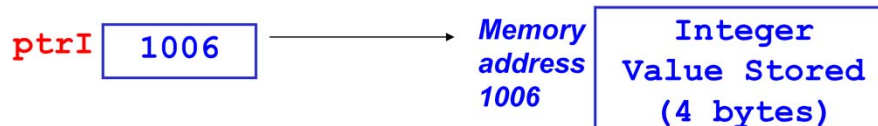
## Pointer Variables: Declaration

- **Pointer variable** – different from the variable **num** (variable of primitive data type such as int, float, char) declared earlier, it stores the **address** of memory location of a data object.
- A **pointer variable** is declared by, for example:

```
int *ptrI;
or int *ptrI;
or int *ptrI;
```



- **ptrI** is a pointer variable. It does **not** store the **value** of the variable. It stores the **address** of the memory which is used for storing an Int value. Diagrammatically,



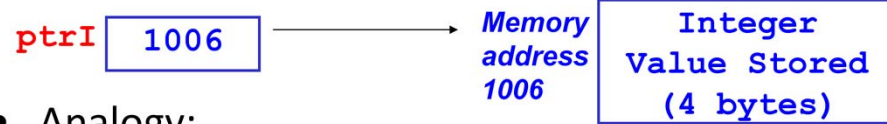
8

### Pointer Variables: Declaration

1. Different from the variables of primitive data types which stores the values directly, we may also have variables which store the addresses of memory locations of some data objects. These variables are called pointers. A *pointer variable* is declared by: **data\_type \*ptr\_name;** where **data\_type** can be any C data type such as **char**, **int**, **float** or **double**. **ptr\_name** is the name of the pointer variable.
2. The **data\_type** is used to indicate the type of data that the variable is pointed to. An asterisk (\*) is used to indicate that the variable is a pointer variable.
3. For example, the statement **int \*ptrI;** declares a pointer variable **ptrI** that points to the address of a memory location that is used to store an **int**.
4. Note that the value of a pointer variable is an **address**, which is different from other variables of primitive data types that store the **data** directly. If we want to retrieve the actual value, we will need to use **indirection operator (\*)**, i.e. **\*ptrI**.



## Pointer Variables: Declaration (Cont'd.)



- Analogy:  
(1) Address on envelope → your home



- (2) Bank account → your saving/money in the bank



9

### Pointer Variables: Declaration

1. Pointer variable is similar to the address written on an envelope which stores the home address, and the actual place can be referred to by the address.
2. It is also similar to a bank account book, which contains the saving information and the bank location that stores the money. The money can be referred to via the bank account.

## Pointer Variables: Declaration (Cont'd.)

**float \*ptrF;**

ptrF is a pointer variable. It stores the **address** of the memory which is used for storing a Float value.

ptrF 2024

Memory  
address  
2024

Float value  
stored (4 bytes)

**char \*ptrC;**

ptrC is a pointer variable. It stores the **address** of the memory which is used for storing a Character value.

ptrC 3024

Memory  
address  
3024

Character value  
stored (1 byte)

10

### Pointer Variables: Declaration

1. The statement **float \*ptrF;** declares a pointer variable **ptrF** that points to the address of a memory location that is used to store a **float**.
2. Similarly, the statement **char \*ptrC;** declares a pointer variable **ptrC** that points to the address of a memory location that is used to store a **char**.
3. When a pointer is declared without initialization, 4 bytes of memory are allocated to the pointer variable. However, no data or address is stored in the memory.

## Pointer Variables: Key Ideas

```
int * ptrI;
```

You need to understand the following 2 concepts:

### (1) ptrI

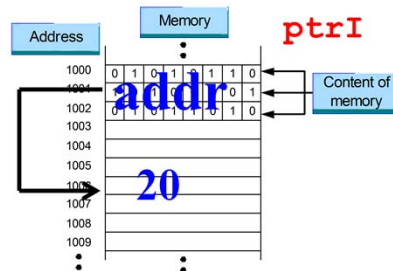
- pointer variable
- the value of the variable (i.e. stored in the variable) is an address

### (2) \*ptrI

- contains the content (or value) of the memory location pointed to by the pointer variable ptrI
- referred to by using the indirection operator (\*), i.e. \*ptrI, \*ptrF, \*ptrC.
- For example: we can assign

```
*ptrI = 20;
```

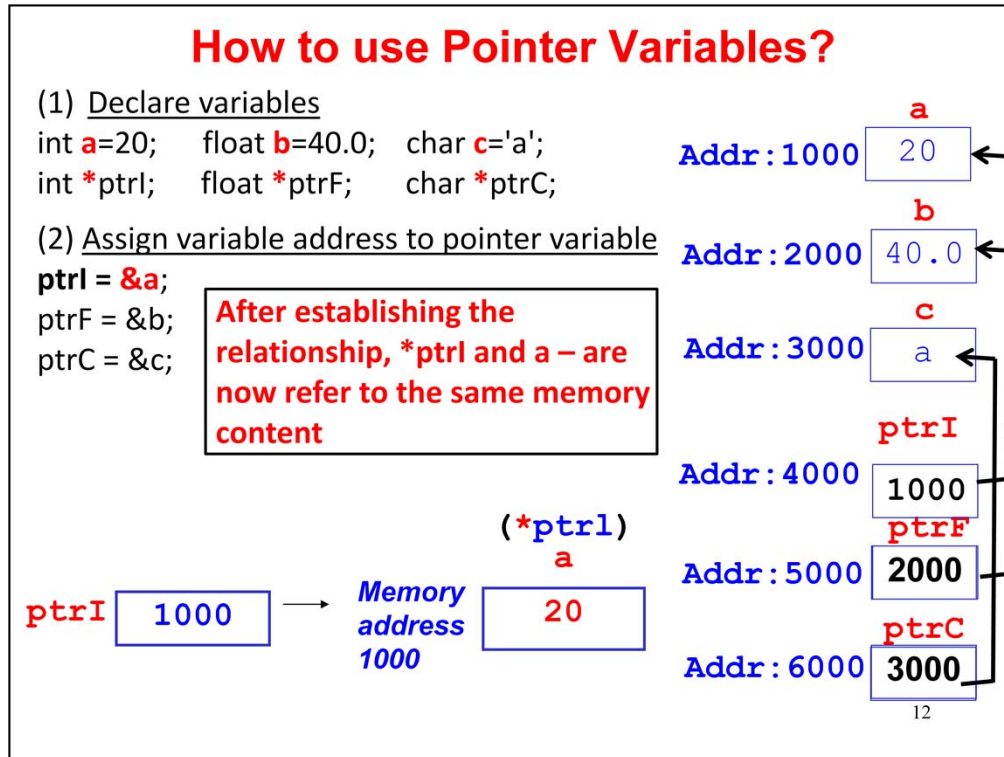
=> the value 20 is stored at the address pointed to by ptrI.



11

### Pointer Variables: Key Ideas

- There are two important concepts related to pointers:
  - The pointer variable (**ptr**) is used to store an address which refers to the location that stores the actual data of the specified data type.
  - The indirection operator (**\*ptr**) can be used to retrieve the actual value pointed to by the pointer variable.
- For example, after declaring the pointer variable, the following assignment statement, **\*ptrI = 20;** will update the memory location pointed to by the pointer variable to 20.
- As such, we can retrieve the actual integer value of 20 referred to by the pointer variable **ptrI** using indirection operator **\*ptrI** which will give the value of 20.



### How to use Pointer Variables?

1. Declare and initialize the variables and pointer variables as follows:  
`int a=20; float b=40.0; char c='a';`  
`int *ptrI; float *ptrF; char *ptrC;`
2. We can then assign variable address to pointer variable as follows:  
`ptrI=&a;`  
`ptrF=&b;`  
`ptrC=&c;`
3. The value of a pointer variable is an address. The pointer variables then point to the memory locations that are used to store the values. The statement **ptrI = &a;** is used to assign the address of the memory location of **a** to the pointer variable **ptrI**. Similarly, we can write the other assignment statements as **ptrF = &b;** and **ptrC = &c;**

Apart from storing an address value in a pointer variable, we can also store the **NULL** value which is defined in `<stdio.h>` in a pointer variable. The pointer is then called a **NULL** pointer. **NULL** is represented in the computer as a series of 0 bits. It refers to the memory location 0. It is common to initialize a pointer to **NULL** in order to avoid it pointing to a random memory location:

```
int *ptr = NULL;
```

## How to use Pointer Variables? (Cont'd.)

```

int a=20;    float b=40.0;  char c='a';
int *ptrI;   float *ptrF;   char *ptrC;
ptrI = &a;   => *ptrI == 20 [same as variable a]
ptrF = &b;    => *ptrF == 40.0 [same as b]
ptrC = &c;    => *ptrC == 'a' [same as c]
  
```

Statement	Operation
int *ptrI	ptrI ? Uninitialized Pointer
ptrI = &a;	<div style="display: flex; align-items: center;"> <div style="text-align: right; padding-right: 10px;">ptrI 1000</div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; right: -10px; top: -5px;">a</div> <div style="position: absolute; right: -10px; bottom: -5px;">20</div> </div> <div style="text-align: left; padding-left: 10px;">Address = 1000</div> </div>
ptrF = &b;	<div style="display: flex; align-items: center;"> <div style="text-align: right; padding-right: 10px;">ptrF 2000</div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; right: -10px; top: -5px;">b</div> <div style="position: absolute; right: -10px; bottom: -5px;">40.0</div> </div> <div style="text-align: left; padding-left: 10px;">Address = 2000</div> </div>
ptrC = &c;	<div style="display: flex; align-items: center;"> <div style="text-align: right; padding-right: 10px;">ptrC 3000</div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; right: -10px; top: -5px;">c</div> <div style="position: absolute; right: -10px; bottom: -5px;">a</div> </div> <div style="text-align: left; padding-left: 10px;">Address = 3000</div> </div>
int *ptr = NULL;	<div style="display: flex; align-items: center;"> <div style="text-align: right; padding-right: 10px;">ptr NULL</div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; right: -10px; top: -5px;">⏚</div> </div> </div>

Addr: 1000

a  
20

Addr: 2000

b  
40.0

Addr: 3000

c  
a

\*ptrI and a –  
now refer to  
the same  
memory  
content

13

### How to use Pointer Variables?

1. After a pointer variable is assigned to point to a data object or variable, we can access the value stored in the variable using *indirection operator* (\*). If the pointer variable is defined as **ptr**, we use the expression **\*ptr** to dereference the pointer to obtain the value stored at the address pointed at by the pointer **ptr**.
2. After the assignment operations, we will have:
  - **ptrI** stores the memory address of the variable **a**;
  - **ptrF** stores the memory address of the variable **b**;
  - **ptrC** stores the memory address of the variable **c**.
3. It implies that:
  - **\*ptrI** and **a** will have the same value 20;
  - **\*ptrF** and **b** will have the same value 40.0;
  - **\*ptrC** and **c** will have the same value 'a'.
4. It means that after the assignment **ptrI = &a**; we will be able to retrieve the value of the variable **a** through either (1) the variable **a** directly; or (2) dereferencing the pointer variable **\*ptrI**. Therefore, we can write programs more flexibly by using pointer variable.

## Pointer Variables – Example 1

```
#include <stdio.h>
int main()
{
    int num = 3; // integer var
    int *ptr;    // pointer var

    ptr = &num; // assignment

    // Question: what will be ptr, *ptr, num?
    printf("num = %d &num = %p\n", num, &num);

    printf("ptr = %p *ptr = %d\n", ptr, *ptr);

}
```

Statement	Operation
ptr = &num;	<p>ptr: 1024 → num: 3 Address = 1024</p>
*ptr = 10;	<p>ptr: 1024 → num: 10 Address = 1024</p>

**Output**

num = ?

&num = ?

ptr = ?

\*ptr = ?

14

### Pointer Variables: Example 1

1. In the program, a pointer variable **ptr** is declared to point to a variable of type **int**. The statement **ptr = &num;** assigns the address of the variable **num** to the pointer variable **ptr**.
2. The statement **printf("ptr = %p, \*ptr = %d\n", ptr, \*ptr);** prints the value (or address value) stored in the pointer variable **ptr**, and the content of the memory location pointed to by the pointer variable. This refers to the same value stored at the variable **num**.

## Pointer Variables – Example 1 (Cont'd.)

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

```
int main()
```

```
{
```

```
    int num = 3; // integer var
```

```
    int * ptr; // pointer var
```



```
    ptr = &num; // assignment
```

```
    // Question: what will be ptr, *ptr, num?
```

```
    printf("num = %d &num = %p\n", num, &num);
```

```
    printf("ptr = %p *ptr = %d\n", ptr, *ptr);
```

```
}
```

Statement	Operation
<code>ptr = &amp;num;</code>	
<code>*ptr = 10;</code>	

### Output

```
num = 3,  
&num = 1024
```

```
ptr = 1024,  
*ptr = 3  
[num and *ptr  
have the same  
value]
```

15

### Pointer Variables: Example 1 [Overview]

1. The primitive type variable **num** stores the value of 3, and the address of the memory location of the variable **num** is 1024.
2. The value stored in the pointer variable (i.e. **ptr**) is 1024, and the value referred to by the pointer variable (i.e. \***ptr**) is 3.
3. After the assignment statement: **ptr = &num**; the pointer variable **ptr** stores the address of the memory location of the variable **num**.
4. Therefore, the values for **ptr** and **&num** are the same (i.e. 1024). And the values for the variable **num** and the dereferencing of the pointer variable \***ptr** are the same (i.e. 3).



## Pointer Variables – Example 1 (Cont'd.)

```
#include <stdio.h>
int main()
{
    int num = 3; // integer var
    int *ptr;    // pointer var
```

```
    ptr = &num;
```

```
    printf("num = %d &num = %p\n", num, &num);
```

```
    printf("ptr = %p *ptr = %d\n", ptr, *ptr);
```



```
    *ptr = 10;
```

```
    // What will be the values for *ptr, num, &num?
```

```
    printf("num = %d &num = %p\n", num, &num);
```

```
    return 0;
```

```
}
```

Statement	Operation
ptr = &num;	
*ptr = 10;	

### Output

```
num = 3
&num = 1024
```

```
ptr = 1024
*ptr = 3
```

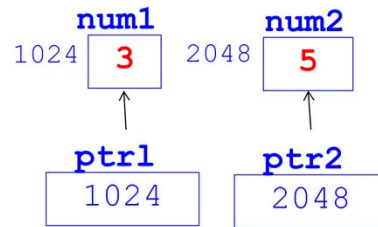
```
num = 10
[*ptr = 10] 16
&num = 1024
```

### Pointer Variables: Example 1

1. The statement **\*ptr = 10;** assigns the value 10 to the variable **num**.
2. Since **ptr** stores the address of **num**, the change of value at this memory location has the same effect as changing the value stored at **num**.
3. Therefore, the value stored at **num** is 10. And **\*ptr** is also 10. There is no change to the address of the memory location of **num** (i.e. 1024).

## Pointer Variables – Example 2

```
/* Example to show the use of pointers */
#include <stdio.h>
int main()
{
    int num1 = 3, num2 = 5; // integer variables
    int *ptr1, *ptr2;       // pointer variables
```



```
ptr1 = &num1; /* put the address of num1 into ptr1 */
// What are the values for num1, *ptr1?
printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);
```

```
ptr2 = &num2; /* put the address of num2 into ptr2 */
// What are the values for num2, *ptr2?
printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);
```

**Output**  
**num1 = 3, \*ptr1 = 3**  
**num2 = 5, \*ptr2 = 5**

17

### Pointer Variables: Example 2

1. In the program, the following statements assign the address of **num1** into **ptr1**, and the address of **num2** into **ptr2**:

**ptr1 = &num1;**

**ptr2 = &num2;**

2. Therefore, we have

**num1 = 3** and **\*ptr1 = 3**; and

**num2 = 5** and **\*ptr2 = 5**.

**Pointer Variables – Example 2 (Cont'd.)**

```

/* increment by 1 the content of the memory
location pointed by ptr1 */
(*ptr1)++;

// What are the values for num1, *ptr1?
printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

```

18

### Pointer Variables: Example 2

1. The statement **(\*ptr1)++**; increments 1 to the content of the memory location pointed to by **ptr1**.
2. Therefore, we have **\*ptr1 = 4**, and **num1 = 4**.

**Pointer Variables – Example 2 (Cont'd.)**

```

/* copy the content of the location pointed by ptr1
into the location pointed by ptr2*/

*ptr2 = *ptr1;

// What are the values for num2, *ptr2?
printf("num2 = %d,*ptr2 = %d\n",num2, *ptr2);

```

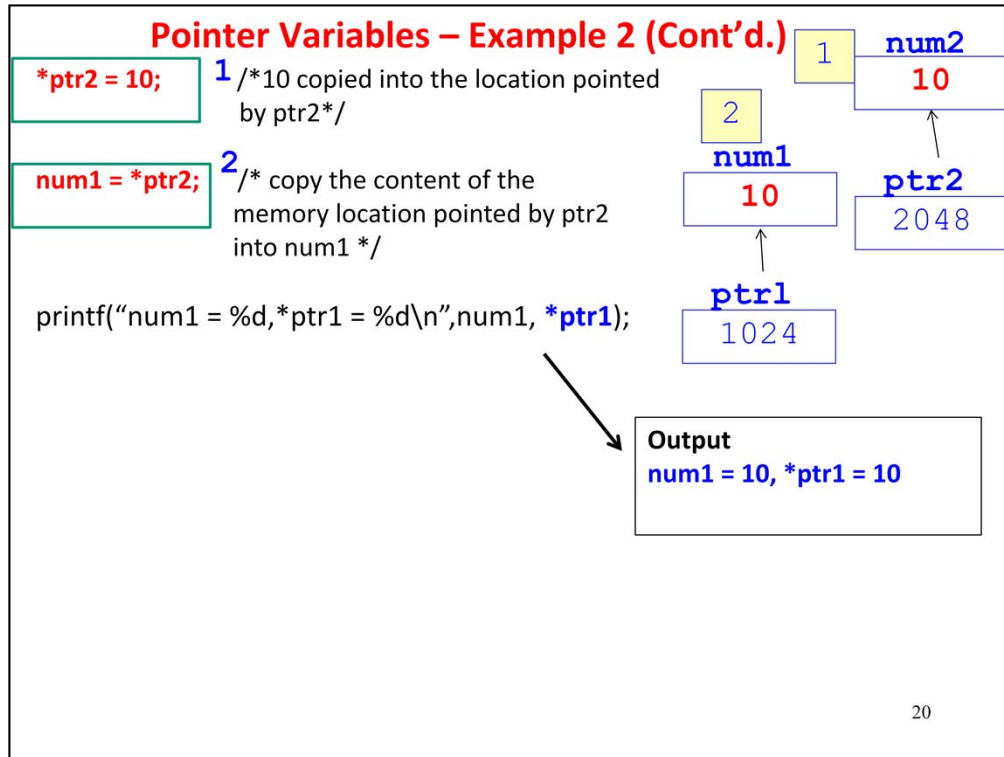
**Output**

**num2 = 4, \*ptr2 = 4**

19

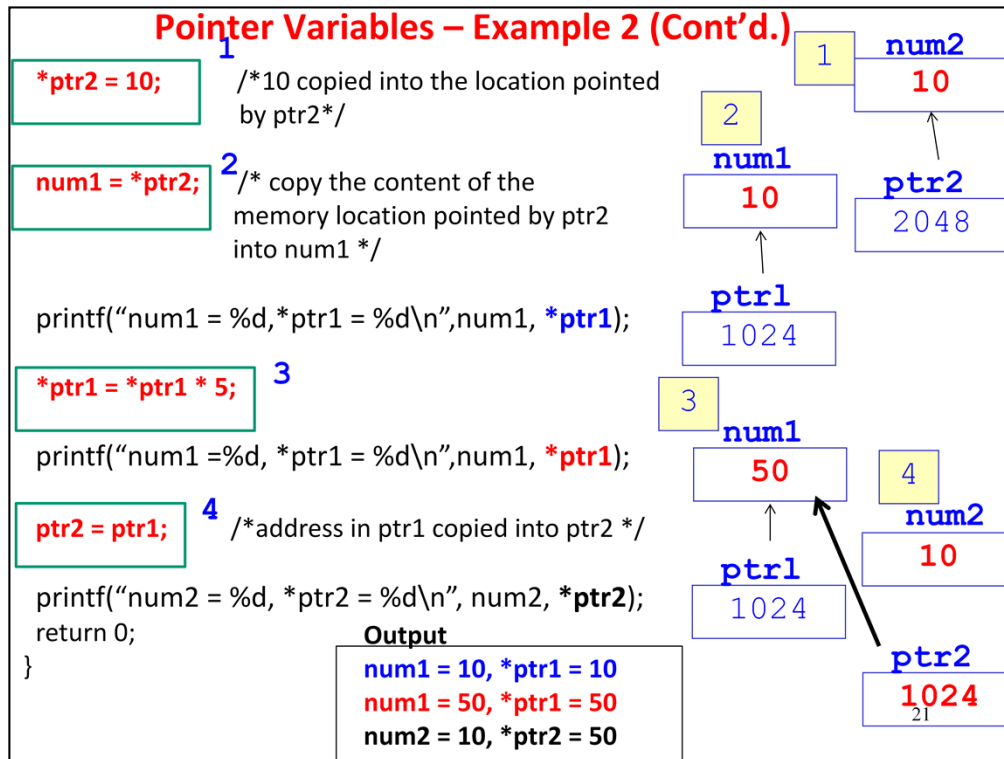
### Pointer Variables: Example 2

1. The statement **\*ptr2 = \*ptr1;** copies the content of the location pointed to by **ptr1** into the location pointed to by **ptr2**.
2. Since **\*ptr1 = 4**, we have **\*ptr2 = 4** and **num2 = 4**.



### Pointer Variables: Example 2

1. The statement **\*ptr2 = 10;** assigns the value 10 to the content of the memory location pointed to by **ptr2**.
2. Therefore, **\*ptr2 = 10**, and **num2 = 10**.
3. The statement **num1 = \*ptr2;** copies the content of the memory location pointed to by **ptr2** into **num1**.
4. Since **\*ptr2 = 10**, **num1 = 10**, we have **num1 = 10**, and **\*ptr1 = 10**.



### Pointer Variables: Example 2

1. In the statement **\*ptr1 = \*ptr1 \* 5;** since **\*ptr1 = 10**, we have **\*ptr1 \* 5 = 50**.
2. The new value 50 is assigned to the content of the memory location pointed to by **ptr1**. Therefore, we have **\*ptr1 = 50**, and **num1 = 50** as well.
3. The last statement **ptr2 = ptr1;** copies the address in **ptr1** into **ptr2**, so that the pointer variable **ptr2** points to the same memory location as **ptr1**. Therefore, we have **\*ptr2 = 50**.
4. However, the value of **num2** is not changed, we have **num2 = 10**.

The concepts of using pointers in the program are summarized as follows:

- **ptr1, ptr2** - They refer to the values (which are memory addresses) stored in the pointer variables **ptr1** and **ptr2**.
- **&ptr1, &ptr2** - They refer to the memory addresses of the variables **ptr1** and **ptr2**.
- **\*ptr1, \*ptr2** - They refer to the values (which are primitive data) whose memory locations are stored in the memory locations of the pointer variables **ptr1** and **ptr2**.

## Using Pointer Variables (within the Same Function): Key Steps

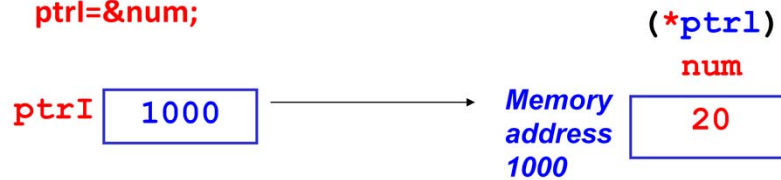
1. Declare variables and pointer variables:

```
int num=20;
```

```
int *ptr1;
```

2. Assign the address of variable to pointer variable:

```
ptr1=&num;
```



**Then you can retrieve the value of the variable num through \*ptr as well ....**

22

### Using Pointer Variables: Key Steps

1. There are two key steps on using pointer variables:

- a) Declare variables and pointer variables:

```
int num=20;
```

```
int *ptr1;
```

- a) Assign the address of variable to pointer variable:

```
ptr1=&num;
```

## Pointers

- Primitive Data Types, Variables and Address Operator
- Pointer Variables
- **Call by Reference**

23

### Pointers

1. Here, we discuss the concept of call by reference.



## Call by Reference

- Parameter passing between functions has two modes:
  - **call by value** [in the last lecture on Function]
  - **call by reference** [to be discussed in this lecture]
- **Call by reference**: the parameter in the function holds the address of the argument variables, i.e. the parameter is a pointer variable.
  - In a **function call**, the **arguments** must be **pointers** (or using address operator as the prefix).  
     E.g. `double x1,y1;`  
         ....  
         `distance(&x1, &y1);`
  - In the **function header's** parameter declaration list, the **parameters** must be prefixed by the **indirection operator** `*`.  
     E.g. `void distance(double *x, double *y)`

24

### Call by Reference

1. Parameter passing between functions can be done either through call by value or call by reference. Call by value has been discussed in the last lecture on functions.
2. In call by reference, parameters hold the addresses of the arguments, i.e. parameters are **pointers**. Therefore, any changes to the values pointed to by the parameters change the arguments. The arguments must be the addresses of variables that are local to the calling function.
3. In a function call, the **arguments** must be **pointers** (or using address operator as the prefix).
4. In the function header, the **parameters** in the parameter declaration list must be prefixed by the **indirection operator**.

## Recap: Call by Value

- **Call by Value** - Communications between a function and the calling body is done through arguments and the return value of a function.

```

#include <stdio.h>
int add1(int);

int main( )
{
    int num = 5;
    num = add1(num); // num – called argument
    printf("The value of num is: %d", num);
    return 0;
}

int add1(int value) // value – called parameter
{
    value++;
    return value;
}

```

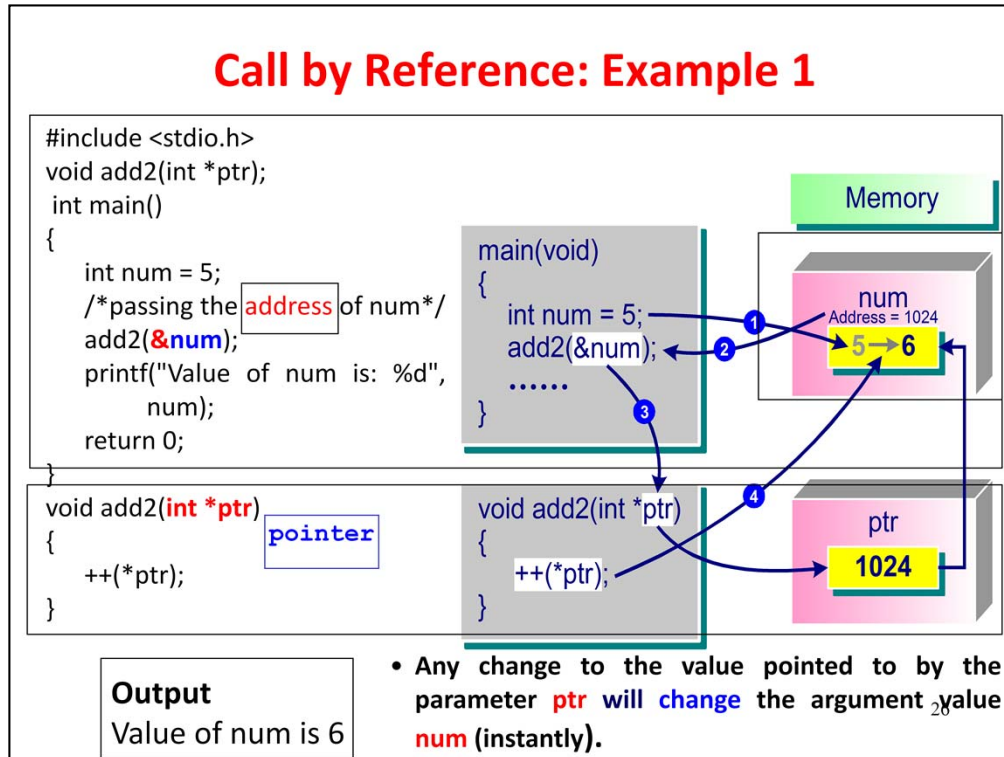
**Output**  
 The value of num is: 6

**num**   5 -> 6

**value**   5 -> 6

### Recap: Call by Value

1. This example illustrated call by value was used in the chapter on Functions.
2. In this example, we pass in the value of the variable **num** as argument from the **main()** function to the parameter **value** of the **add1()** function.



#### Call by Reference: Example 1

- In the program, the variable **num** is initially assigned with a value 5 in **main()**.
- The address of the variable **num** is then passed as an argument to the function **add2()** (step 2) and stored in the parameter **ptr** in the function (step 3).
- In the function **add2()**, the value of the memory location pointed to by the variable **ptr** (i.e. **num**) is then incremented by 1 (step 4). It implies that the value stored in the variable **num** becomes 6. When the function ends, the control is then returned to the calling **main()** function. Therefore, when **num** is printed, the value 6 is displayed on the screen.
- In this example, note that the parameter variable **ptr** in **add2()** is used to store the address of the variable **num** in **main()**. After passing the variable address of **num** into the parameter variable **ptr**, all the operations on **ptr** in the function **add2()** will update the content of the variable **num** indirectly.

## Call by Reference: Key Steps

1. In the **function definition**, the parameter must be prefixed by **indirection operator \***:

```
add2( ) function header: void add2( int *ptr ) { ...}
```

2. In the **calling function**, the arguments must be pointers (or using **address** operator as the prefix):

```
main/other calling function: add2( &num );
```

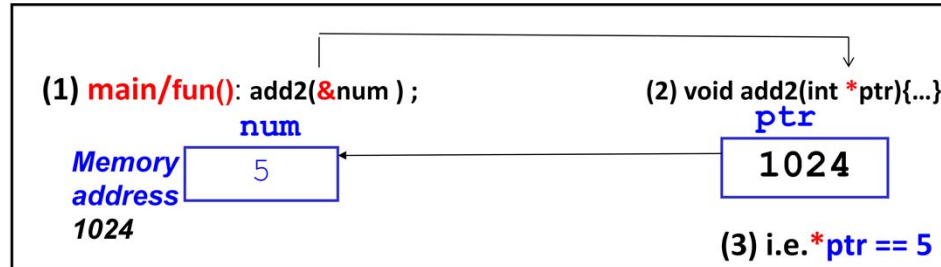
27

### Call by Reference: Key Steps

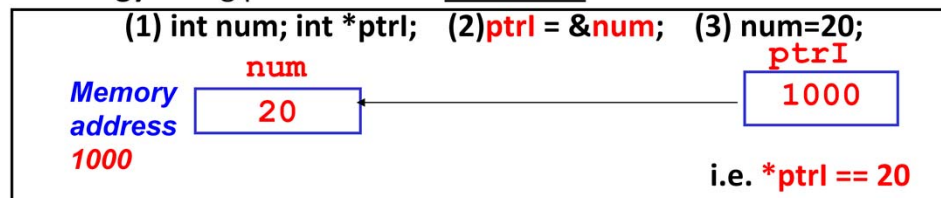
1. There are two key steps when using call by reference:
  - a) In the function, the parameter must be prefixed by the indirection operator:  
**void add2(int \*ptr);**
  - b) In the calling function (e.g. **main()**), the arguments must be pointers (or using address operator as the prefix): **add2(&num);**

## Call by Reference: Analogy

Communications between 2 functions:



**Analogy:** using pointer within a function:



28

### Call by Reference: Analogy

1. Using call by reference via pointer is very similar to that of using pointers in a function. When using pointers in a function:
  - a) We first declare the variable and pointer variable: `int num; int *ptrI;`
  - b) Then, assign the address of the variable **num** to **ptrI**: `ptrI = &num;`
2. Therefore, when the variable **num** is updated to 20: **num** is 20; **\*ptrI** is also 20;

## Call by Reference – Example 2

```
#include<stdio.h>
void function1 (int a, int *b); void function2 (int c, int *d);
void function3 (int h, int *k);
int main() {
    int x, y;
    x = 5; y = 5;
    function1(x, &y);
    return 0;
}
void function1(int a, int *b) {
    *b = *b + a;
    function2(a, b);
}
void function2(int c, int *d) {
    *d = *d * c;
    function3(c, d);
}
void function3(int h, int *k) {
    *k = *k - h;
}
```

Diagram annotations: A box labeled "address" points to the `&y` argument in `function1`. Three boxes labeled "pointer" point to the `*b`, `*d`, and `*k` parameters in their respective function definitions.

Comments on the right side of the code:

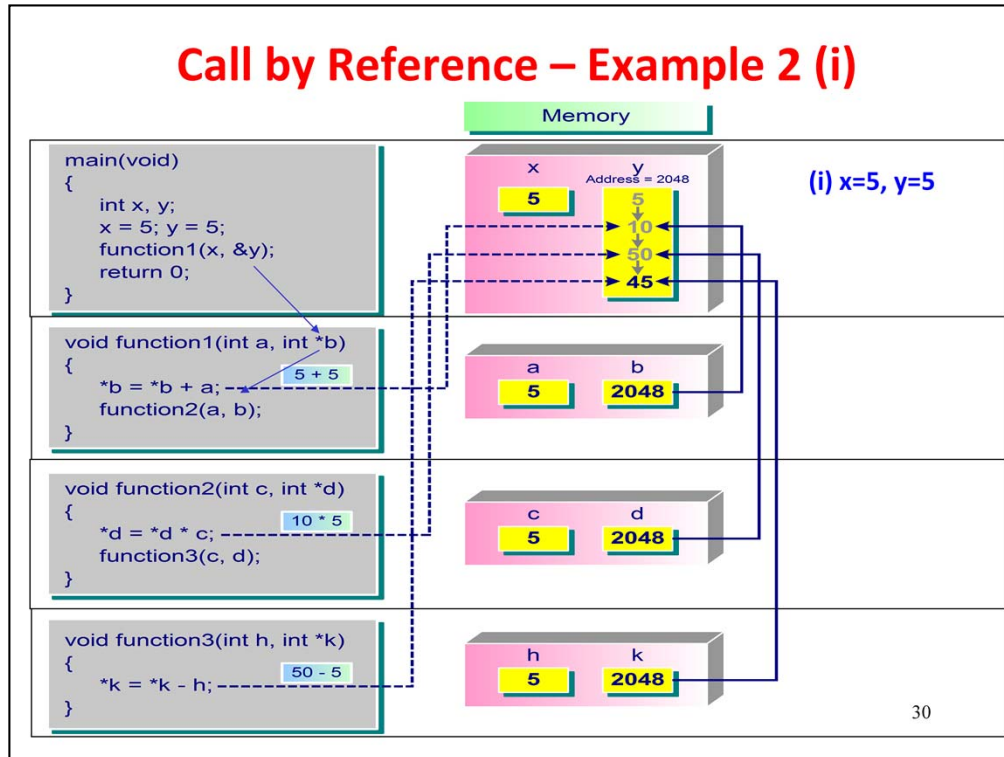
- `/* (i) */` next to `function1(x, &y);`
- `/* (x) */` next to `function1(x, &y);`
- `/* (ii) */` next to `void function1(int a, int *b)`
- `/* (iii) */` next to `*b = *b + a;`
- `/* (ix) */` next to `function2(a, b);`
- `/* (iv) */` next to `void function2(int c, int *d)`
- `/* (v) */` next to `*d = *d * c;`
- `/* (viii) */` next to `function3(c, d);`
- `/* (vi) */` next to `void function3(int h, int *k)`
- `/* (vii) */` next to `*k = *k - h;`

29

### Call by Reference: Example 2

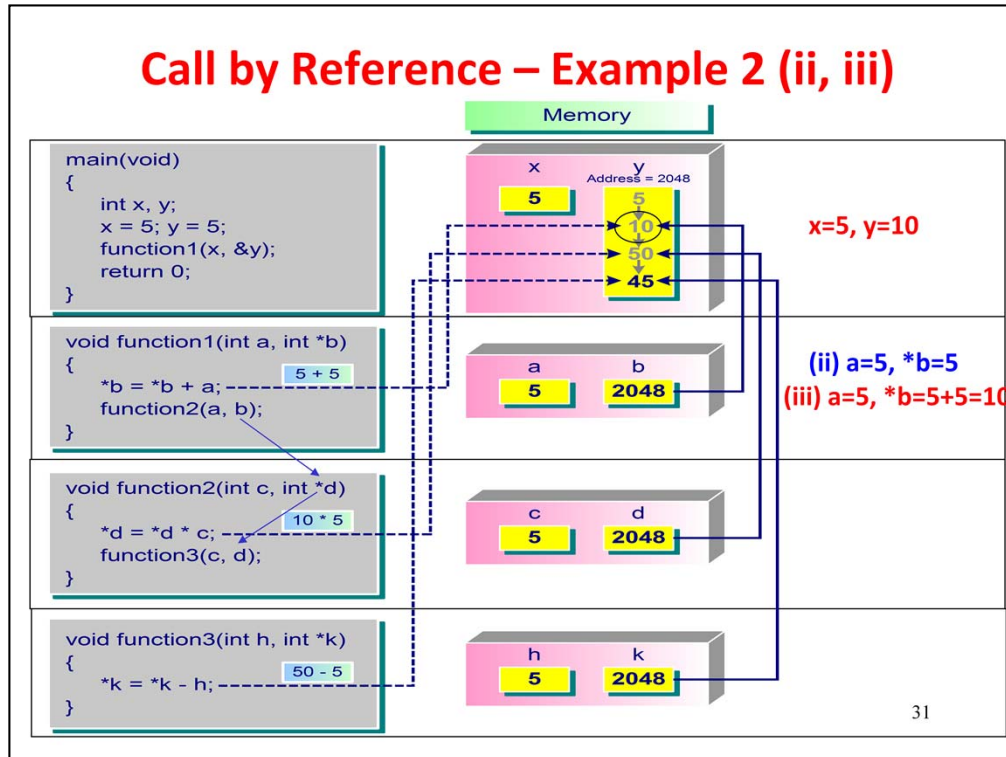
- In the function definitions, we have defined the following three functions:
 

```
void function1(int a, int *b)
void function2(int c, int *d)
void function3(int h, int *k)
```
- The parameters **a**, **c** and **h** are passed into the functions using call by value, whereas the parameters **b**, **d** and **k** are passed into the functions using call by reference, i.e. addresses are passed to the functions instead of actual values.
- In fact, the memory contents of these parameters contain the address of the variable **y** in the **main()** function. Any changes to the dereferenced pointers such as **\*b**, **\*d** and **\*k** refer indirectly to the changes to the contents stored in the memory location of the variable **y**.



#### Call by Reference: Example 2

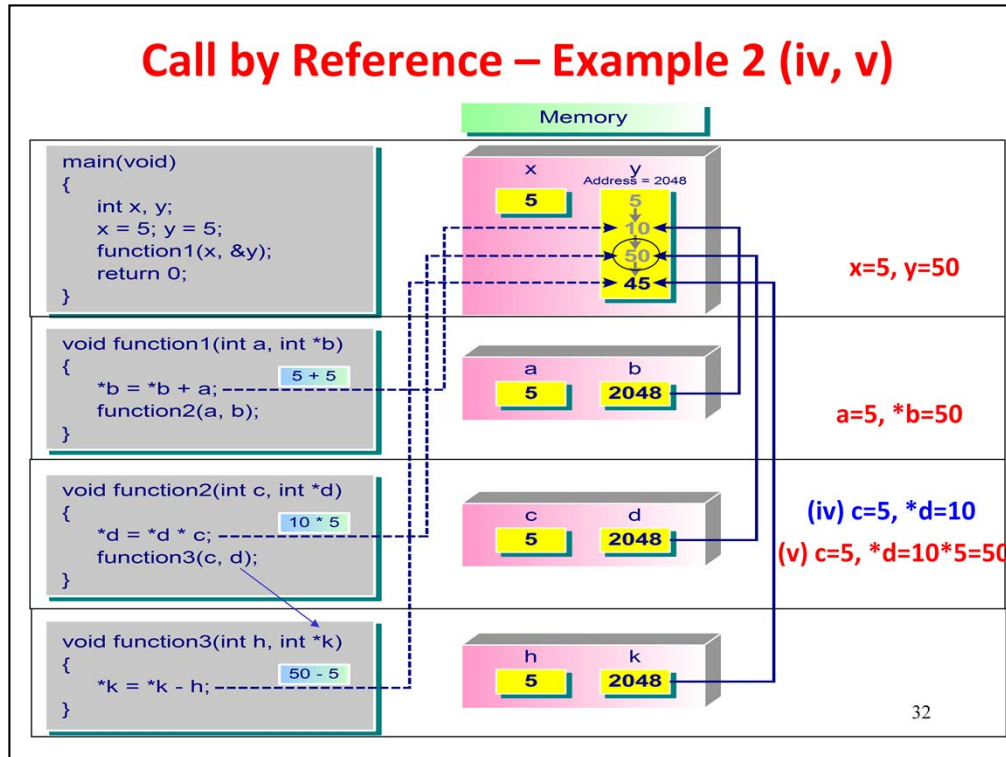
1. When the program starts execution, in the **main()** function, memory locations are allocated to the variables **x** and **y** accordingly. The two variables are assigned with the value of 5. Therefore, the memory locations of the variables **x** and **y** store the value of 5 directly.
2. The **main()** function then calls **function1()** by passing the value of **x** (i.e. 5) and the address of **y** (i.e. 2048) to the corresponding parameters **a** and **b**.
3. The mode of parameter passing for **a** is call by value, and for **b** is call by reference. As such, the parameter **b** refers to the memory location of **y** in the **main()** function.



#### Call by Reference: Example 2

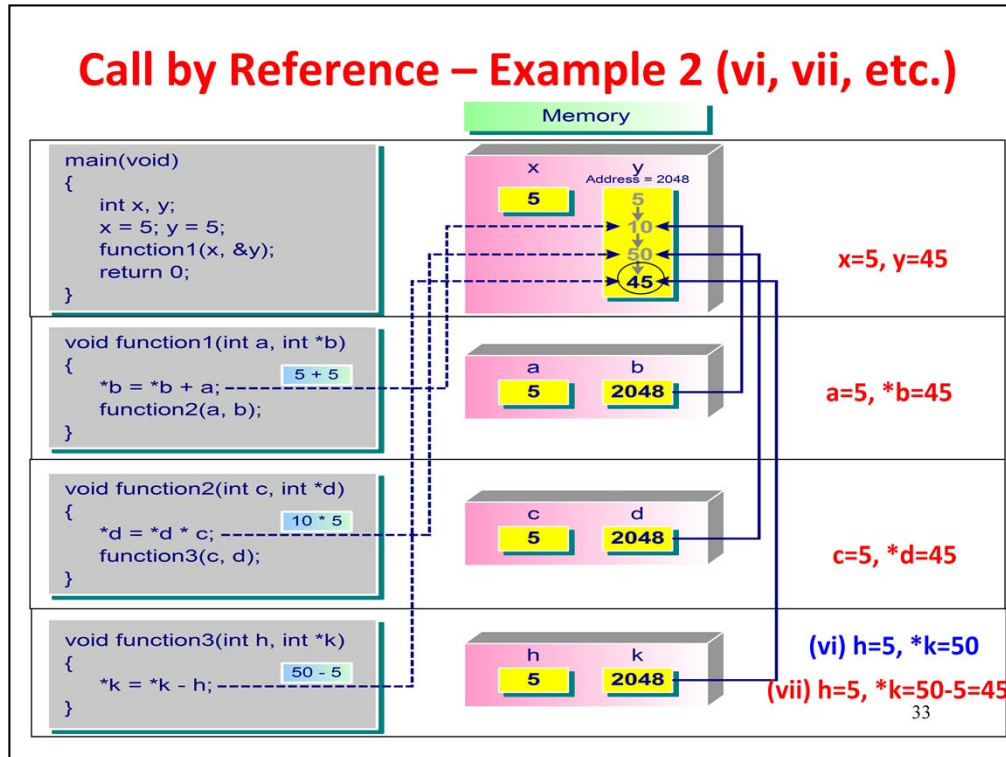
1. When **function1()** is executed, the statement **\*b = \*b + a;** will update the value of **\*b = 5 + 5 = 10**; As the pointer variable **b** refers to the location of the variable **y** in the **main()** function, the update in fact is carried out at the memory location of **y**.
2. Therefore, the value of **y = 10** (in **main()**), and the value of **\*b = 10** (in **function1()**). There is no change in the value of the variable **a** which is 5.
3. After that, **function1()** calls **function2()** by passing in the values of **a** and **b** into the parameters **c** and **d** respectively.





#### Call by Reference: Example 2

1. When **function2()** is executed, the statement **\*d = \*d \* c;** will update the value of **\*d = 10 \* 5 = 50**; As the pointer variable **d** contains the same address value as **b** in **function1()**, it also refers to the location of the variable **y** in the **main()** function. The update in fact is carried out at the memory location of **y**.
2. Therefore, the value of **y** is also 50 (in **main()**), and the value of **\*d = 50** (in **function2()**). There is no change in the value of the variable **c** which is 5.
3. After that, **function2()** calls **function3()** by passing in the values of **c** and **d**.



#### Call by Reference: Example 2

1. When **function3()** is executed, the statement **\*k = \*k - h;** will update the value of **\*k = 50 - 5 = 45**; As the pointer variable **k** contains the same address value as **d** in **function2()**, it also refers to the location of the variable **y** in the **main()** function. The update in fact is carried out at the memory location of **y**.
2. Therefore, the value of **y** is also 45 (in **main()**), and the value of **\*k = 45** (in **function3()**). There is no change in the value of the variable **h** which is 5.
3. After that, **function3()** finishes the execution and terminates. The control passes back to **function2()** for execution and then terminates, which in turn passes back to **function1()**, and then terminates and returns to the **main()** function.

<b>Call by Reference – Example 2</b>									
	<b>x</b>	<b>y</b>	a	<b>*b</b>	c	<b>*d</b>	h	<b>*k</b>	remarks
(i)	5	5	-	-	-	-	-	-	in main
(ii)	5	5	5	5	-	-	-	-	in fn 1
(iii)	5	10	5	10	-	-	-	-	in fn 1
(iv)	5	10	5	10	5	10	-	-	in fn 2
(v)	5	50	5	50	5	50	-	-	in fn 2
(vi)	5	50	5	50	5	50	5	50	in fn 3
(vii)	5	45	5	45	5	45	5	45	in fn 3
(viii)	5	45	5	45	5	45	-	-	return to fn 2
(ix)	5	45	5	45	-	-	-	-	return to fn 1
(x)	5	45	-	-	-	-	-	-	return to main <sup>4</sup>

#### Call by Reference: Example 2

1. The values for each variable and parameter for each function in the program are summarized in the table.
2. Note that the value of the variable **x** in the **main()** function is not changed throughout program execution, while the value of variable **y** is changed after each function call.

## When to Use Call by Reference

When to use call by reference:

- (1) When you need to pass more than one value back from a function.
- (2) When using call by value will result in a large piece of information being copied to the formal parameter, for efficiency reason, for example, passing large arrays or structure records.

35

### When to Use Call by Reference

1. Generally, call by reference is used in the following situations:
  - a) When we need to pass more than one value back from a function, or
  - b) In the case that when we use call by value, it will result in a large piece of information being copied to the parameter. This could happen when we pass a large array size or structure record.

## Double Indirection

```

#include <stdio.h>
int main()
{
    int a=2;
    int *p;
    int **pp; ← double indirection

    p = &a;
    pp = &p;
    a++;
    printf("a = %d, *p = %d, **pp = %d\n", a, *p, **pp);
    return 0;
}

```

**Output**  
a = 3  
\*p = 3  
\*\*pp = 3

**Note:** it could also be \*\*\*ppp, etc. The idea remains the same.

36

### Double Indirection

1. We have seen examples on using indirection operator. Double indirection is also quite commonly used in C programming.
2. In the program, **p** is a pointer variable. A variable can also be declared as **int \*\*pp**; using double indirection. **pp** is also a pointer variable.
3. After the first two assignment statements: **p = &a**; and **pp = &p**; the pointer variable **pp** will be used to store an address of another pointer variable (i.e. **p**) (i.e. it points to another pointer variable), which will in turn store the address of a variable (i.e. **a**) of the corresponding primitive type.
4. Therefore, after the variable declaration and assignment, we will have the output:

```

a=3;
*p=3;
**p=3;

```

**Thank you !!!**



37