

4 Pointers

1

Pointers

1. In this lecture, we discuss Pointers in C.

Why Learning Pointers?

- 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.
- In C, pointers can be used in many ways. These include 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.
- In this lecture, we discuss the concepts of pointers including address operator, pointer variables and call by reference.

2

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. These include 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

3

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);
}
```

Printing the value of the variable

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

Output

num = 5,

Address: 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009

Memory: 0 1 0 1 0 1 0 1, 1 0 1 0 1 0 1 0, 0 1 0 1 0 1 0 1, ...

num → Content of memory

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

4

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

→ **Output**
 num = 5,
10
 num = 10,

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

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

The diagram shows a memory table with addresses from 1000 to 1009. At address 1002, the value 10 is stored. A box labeled 'num' points to this memory location, and another box labeled 'Content of memory' also points to it. The binary representation of 10 (01010) is shown in the memory cells.

Address	Memory
1000	0 1 0 1 0
1001	1 0 1 0 1
1002	0 1 0 1 0
1003	0 1 0 1 0
1004	
1005	
1006	
1007	
1008	
1009	

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**

6

Address Operator

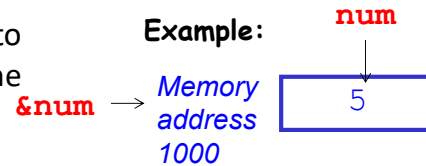
1. The address of a variable can be obtained by the *address operator (&)*. In the program, we can print the address of the variable **num** (i.e. **&num**). To do this, we need to use **%p** in the conversion specifier in the control string of the **printf()** statement: **printf("num = %d, &num = %p\n", 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=5;
```

(1) num

- It is a variable of data type `int` and 4 bytes of memory are allocated.
- Its memory location is used to store the integer value of the variable.



(2) &num

- It refers to the memory address of the variable which is used to store the `int` value of the variable.

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

7

Primitive Variables: Key Ideas

1. There are two important ideas related to primitive variables in the above example:
 - a) First, the primitive variable (**num**) stores a variable value of data type **int**.
 - b) Second, 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

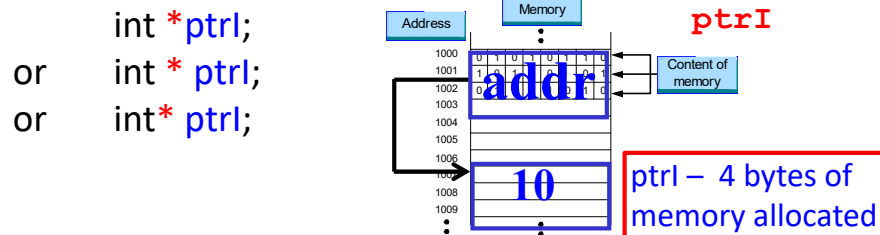
8

Pointers

1. Here, we discuss pointer variables.

Pointer Variables: Declaration

- **Pointer variable** – different from the primitive 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:



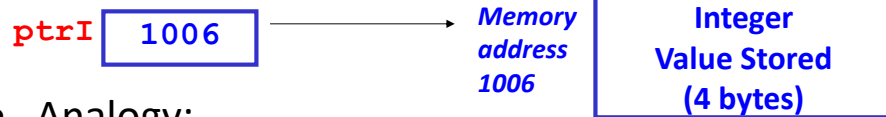
- **ptr1** is a pointer variable. It does **not** store the value of the variable. It stores the address of the memory which is used for storing an Int value.

9

Pointer Variables: Declaration

1. Different from the variables of primitive data types which store the values directly, we may also have variables which store the addresses of memory locations of some data objects. These variables are called **pointer variables**.
2. 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.
3. 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.
4. For example, the statement **int *ptr1;** declares a pointer variable **ptr1** that points to the address of a memory location that is used to store an **int**.
5. 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. ***ptr1**.

Pointer Variables: Analogy



- Analogy:

(1) Address on envelope → your home



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



10

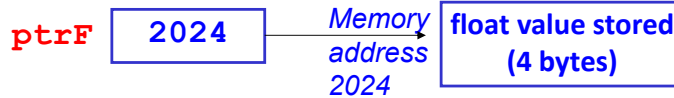
Pointer Variables: Analogy

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 Examples

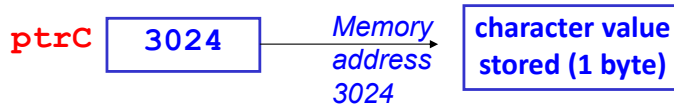
float *ptrF;

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



char *ptrC;

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



11

Pointer Variables: Declaration Examples

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

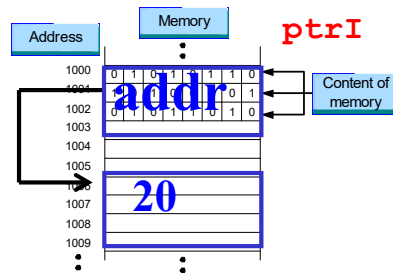
```
int * ptrl;
```

(1) ptrl

- Pointer variable (4 bytes of memory).
- The value of the variable (i.e. stored in the variable) is an **address**.

(2) *ptrl

- Contains the **content (or value)** of the **memory location** pointed to by the pointer variable ptrl.
- The value is referred to by using the **indirection operator (*)**, i.e. *ptrl.
- For example: we can assign
***ptrl = 20;**
 => the value 20 is stored at the address pointed to by ptrl.



12

Pointer Variables: Key Ideas

1. There are two important concepts related to pointers:
 - a) 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.
 - b) The indirection operator (***ptr**) can be used to retrieve the actual value pointed to by the pointer variable.
2. For example, after declaring the pointer variable, the following assignment statement, ***ptrl = 20;** will update the memory location pointed to by the pointer variable to 20.
3. As such, we can retrieve the actual integer value of 20 referred to by the pointer variable **ptrl** using indirection operator ***ptrl** which will give the value of 20.

How to use Pointer Variables?

- **Declare variables**

```
int a=20;   float b=40.0;  char c='a';
int *ptrI;  float *ptrF;   char *ptrC;
```

- After declaration, memories will be allocated for each primitive variable according to its data type.
- For each pointer variable, 4 bytes of memory will be allocated.

Addr: 1000	a 20
Addr: 2000	b 40.0
Addr: 3000	c a
Addr: 4000	ptrI ?
Addr: 5000	ptrF ?
Addr: 6000	ptrC ?

13

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;
```

3. 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]
```

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

Similarly,

***ptrF and b==40.0**

***ptrC and c =='a'**

Statement	Operation
int *ptrI	ptrI ? Uninitialized Pointer
ptrI = &a;	ptrI 1000 → a 20 Address = 1000
ptrF = &b;	ptrF 2000 → b 40.0 Address = 2000
ptrC = &c;	ptrC 3000 → c a Address = 3000
int *ptr = NULL;	ptr NULL

14

How to use Pointer Variables?

1. We can assign variable address to pointer variable as follows:

```
ptrI=&a;
```

```
ptrF=&b;
```

```
ptrC=&c;
```

2. 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;**
3. 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**.
4. 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**.
5. It implies that:

- ***ptrI** and **a** will have the same value of 20;
 - ***ptrF** and **b** will have the same value of 40.0;
 - ***ptrC** and **c** will have the same value of 'a'.
2. 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);
```

```
15 }
```

Note: num and *ptr have the same value

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

Output

num = 3, **&num** = 1024

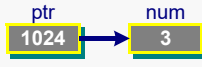
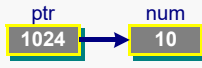
ptr = 1024, ***ptr** = 3

Pointer Variables: Example 1

1. In the program, the primitive type variable **num** stores the value of 3, and the address of the memory location of the variable **num** is 1024.
2. A pointer variable **ptr** is declared to point to a variable of type **int**. The statement **ptr = #** assigns the address of the variable **num** to the pointer variable **ptr**.
3. The first `printf` statement `printf("num = %d, &num = %p\n", num, &num);` prints the value of **num**, and the address of the variable **num**.
4. Therefore, it prints 3 for the value of **num**, and 1024 for the address of **num**.
5. The second `printf` 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.
6. 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. These values are printed on the screen.
7. As can be seen, 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
```

Statement	Operation
ptr = #	
*ptr = 10;	

```
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: *ptr, num, &num?
```

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

```
return 0;
```

```
16 }
```

Output

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

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

```
num = 10, &num = 1024
```

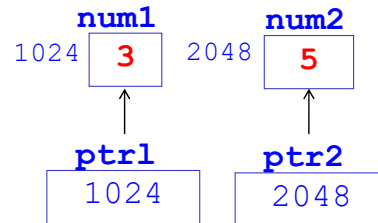
```
[*ptr = 10]
```

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);
```

Output

```
num1 = 3, *ptr1 = 3
num2 = 5, *ptr2 = 5
```

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

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.
```

These values are printed on the screen.

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);

```

Output
num1 = 4, *ptr1 = 4

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**. The values are then printed on the screen.

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);

```

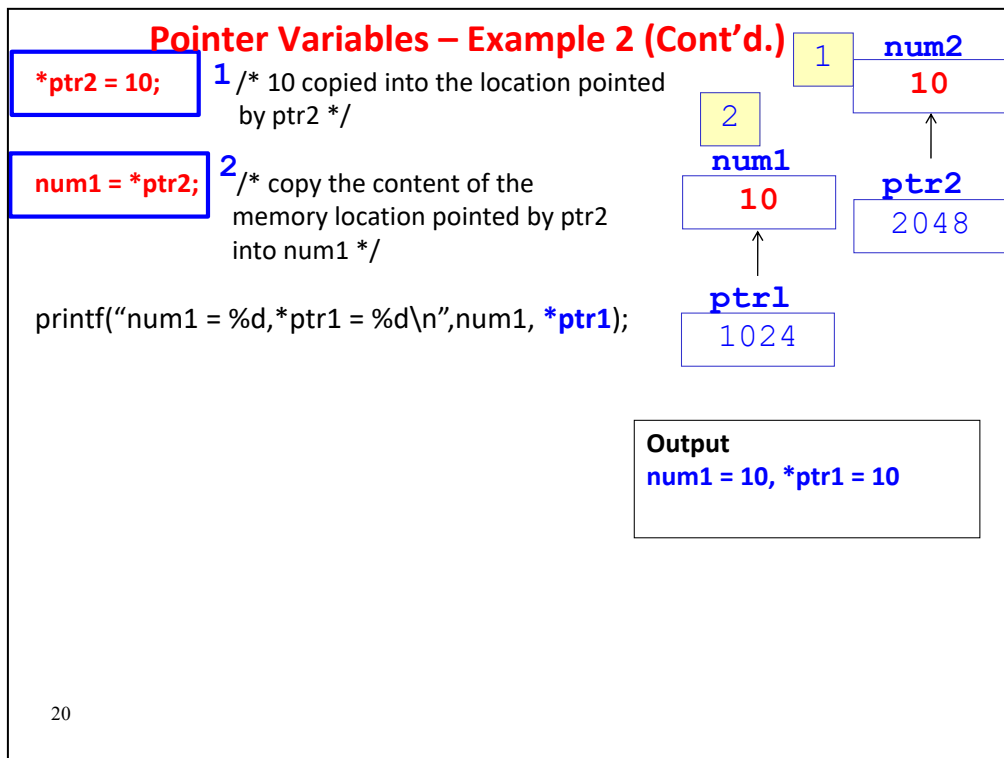
Output

num2 = 4, *ptr2 = 4

19

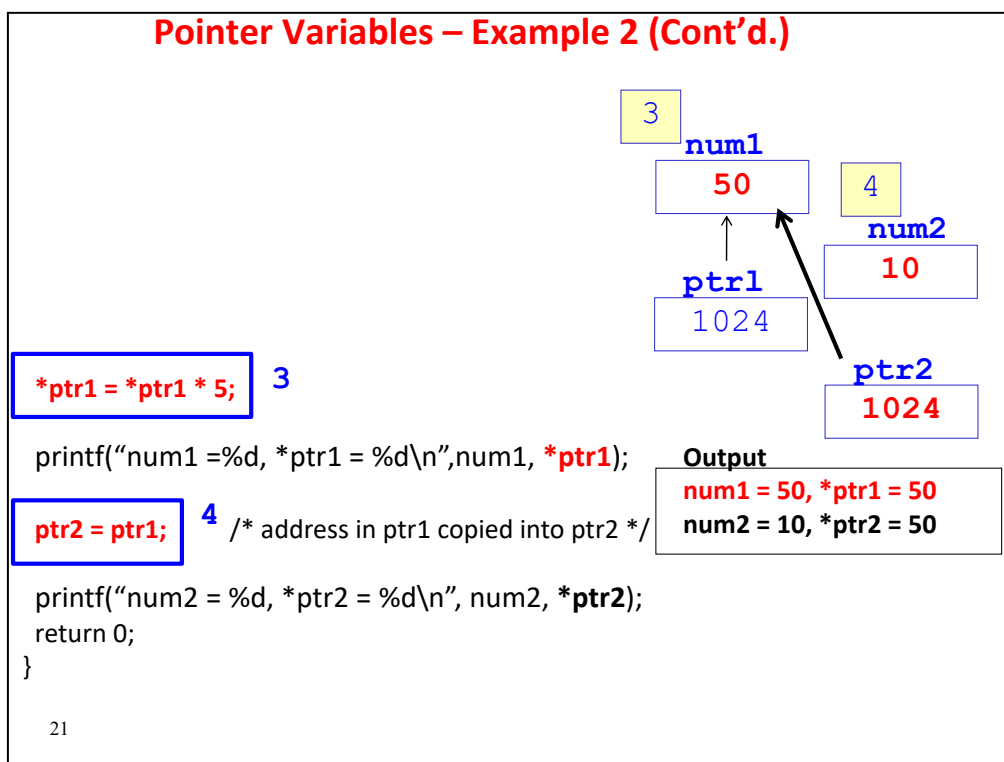
Pointer Variables: Example 2

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**. The values are then printed on the screen.



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**. These values are then printed on the screen.



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**.
3. Therefore, we have ***ptr1 = 50**, and **num1 = 50**. These values are then printed on the screen.
4. 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**.
5. However, the value of **num2** is not changed, we have **num2 = 10**. The values of **num2 = 10** and ***ptr2 = 50** are printed on the screen.
6. 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**.

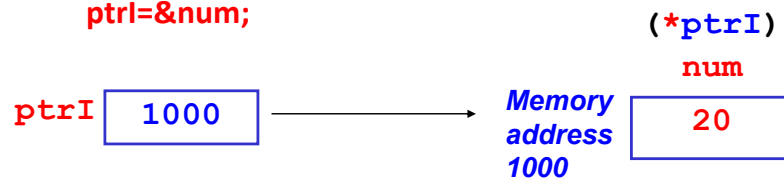
Using Pointer Variables (within the Same Function): Key Steps

1. Declare variables and pointer variables:

```
int num=20;
int *ptrI;
```

2. Assign the address of variable to pointer variable:

```
ptrI=&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) First, declare variables and pointer variables, such as


```
int num=20;
int *ptrI;
```
 - b) Second, assign the address of variable to pointer variable, that is


```
ptrI=&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** [discussed in the last lecture on Functions]
 - **call by reference** [to be discussed in this lecture]
- **Call by reference**: the parameter in the function holds the address of the argument variable, i.e., the parameter is a pointer variable. Therefore,
 - 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)`
 - In the **function call**, the **arguments** must be **pointers** (or using the address operator as the prefix).
 E.g. `distance(&x1, &y1);`

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 chapter 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). For example, `distance(&x1, &y1);`
4. In the function header, the **parameters** in the parameter declaration list must be prefixed by the **indirection operator**. For example, `void distance(double *x, double *y);`

Recap: Call by Value

- **Call by Value** – The communication 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;
}

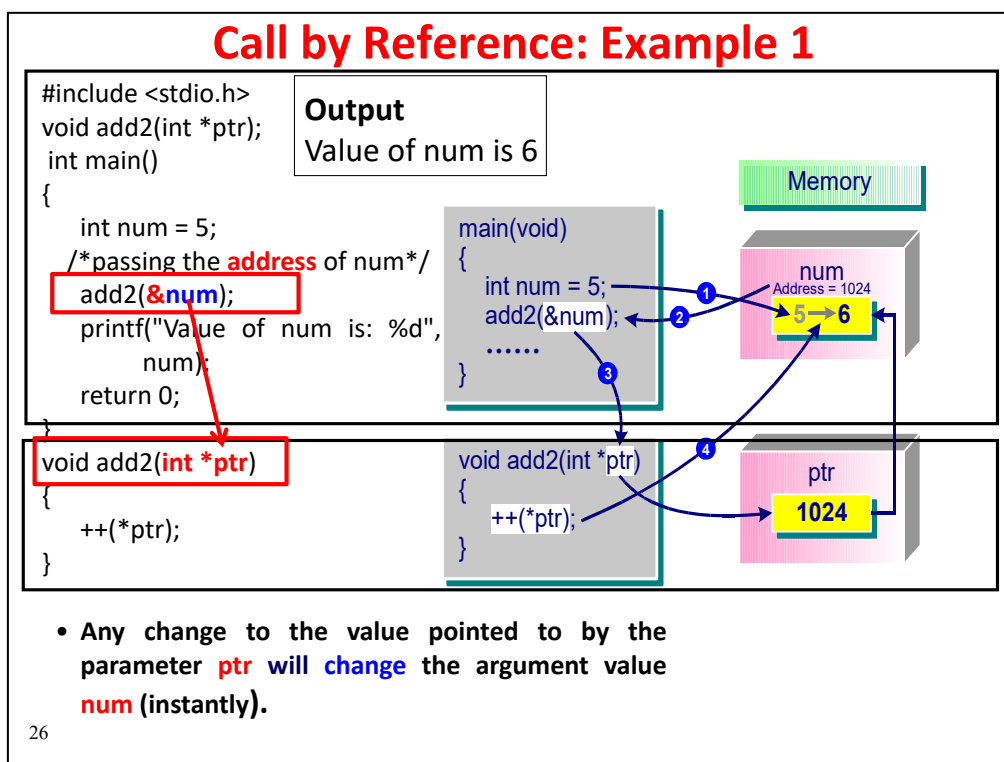
```

Output
The value of num is: 6

25

Recap: Call by Value

1. This example illustrated call by value which was discussed 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 function **add1()**. The value is then used locally for processing in the function **add1()**.



Call by Reference: Example 1

1. In the program, the variable **num** is initially assigned with a value 5 in **main()**.
2. The address of the variable **num** is then passed as an argument to the function **add2()** (in **Step 2**) and stored in the parameter **ptr** in the function (in **Step 3**).
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 (in **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.
4. 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 function **add2()** will update the content of the variable **num** (which is in **main()**) indirectly.

Call by Reference: Key Steps

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

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

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

```
main( ): int num; add2( &num );
```

27

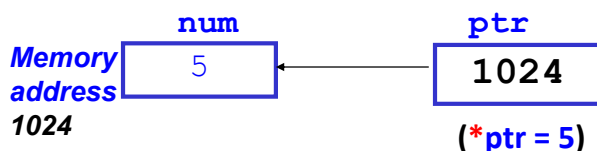
Call by Reference: Key Steps

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

Call by Reference: Analogy

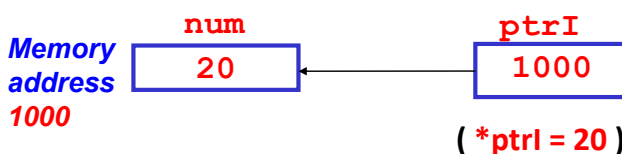
Communications between 2 functions: Call by Reference

(1) **main()**: `int num; add2(&num) ;` (2) `void add2(int *ptr){...}`



Analogy: using pointer within a function:

(1) `int num; int *ptrI;` (2) `ptrI = #`



28

Call by Reference: Analogy

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

- address**: Points to `&y` in `function1(x, &y);`
- pointer**: Points to `*b` in `function1`, `b` in `function2`, `d` in `function3`, and `*k` in `function3`.

Comments on the right side of the code:

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

29

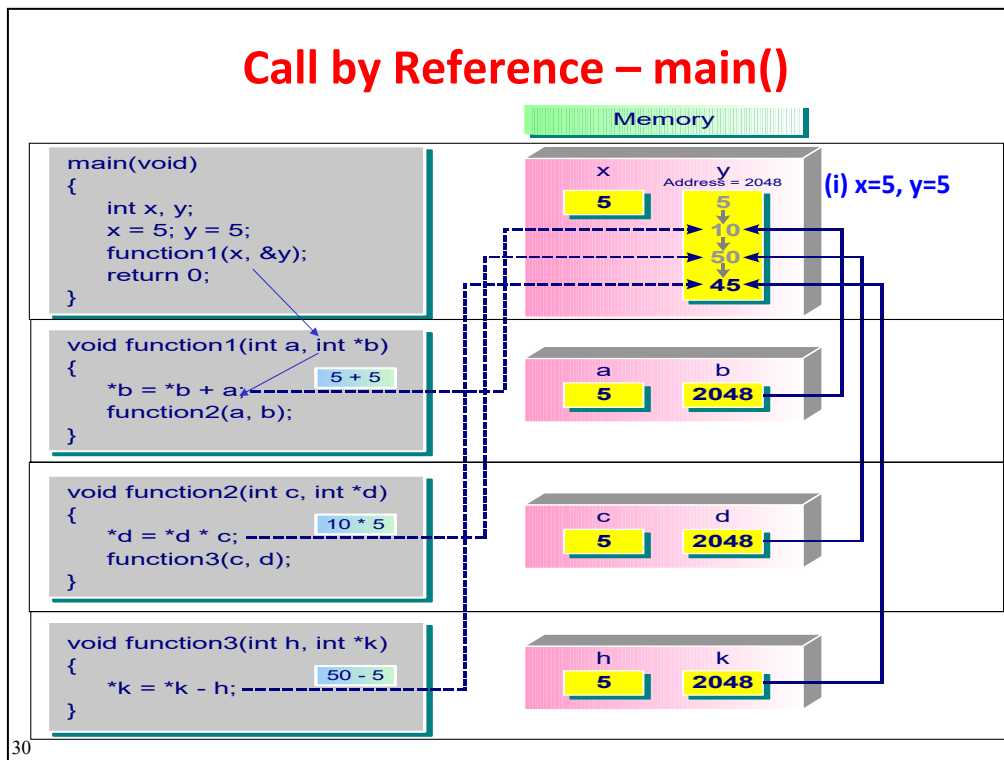
Call by Reference: Example 2

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


```

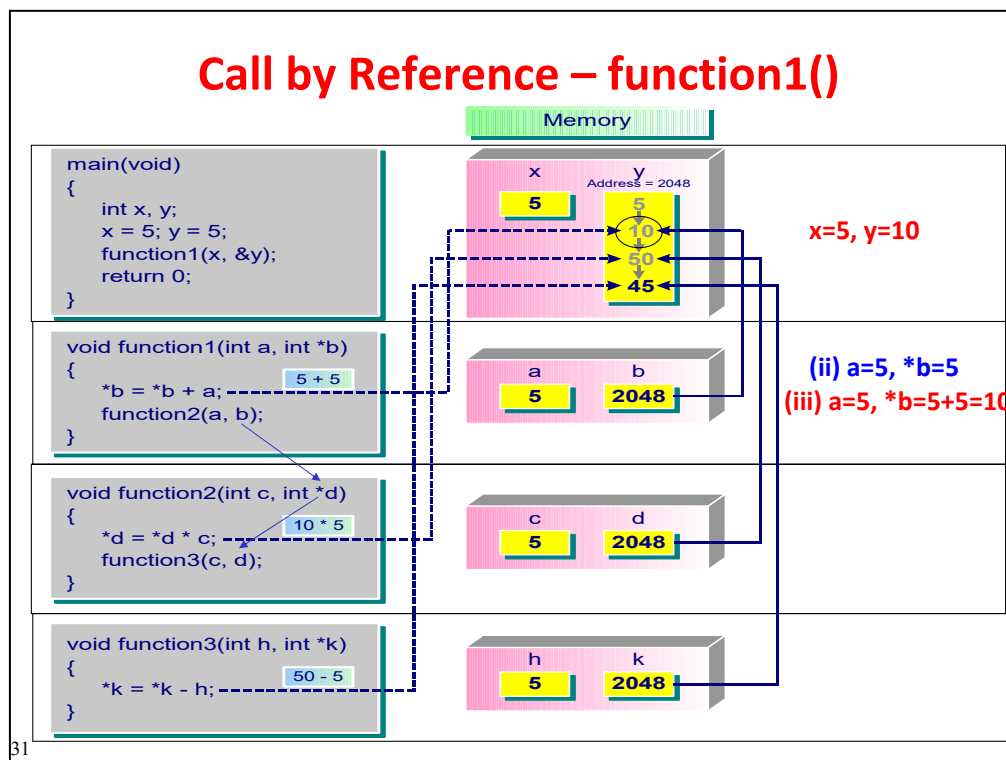
void function1(int a, int *b)
void function2(int c, int *d)
void function3(int h, int *k)

```
- 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**.



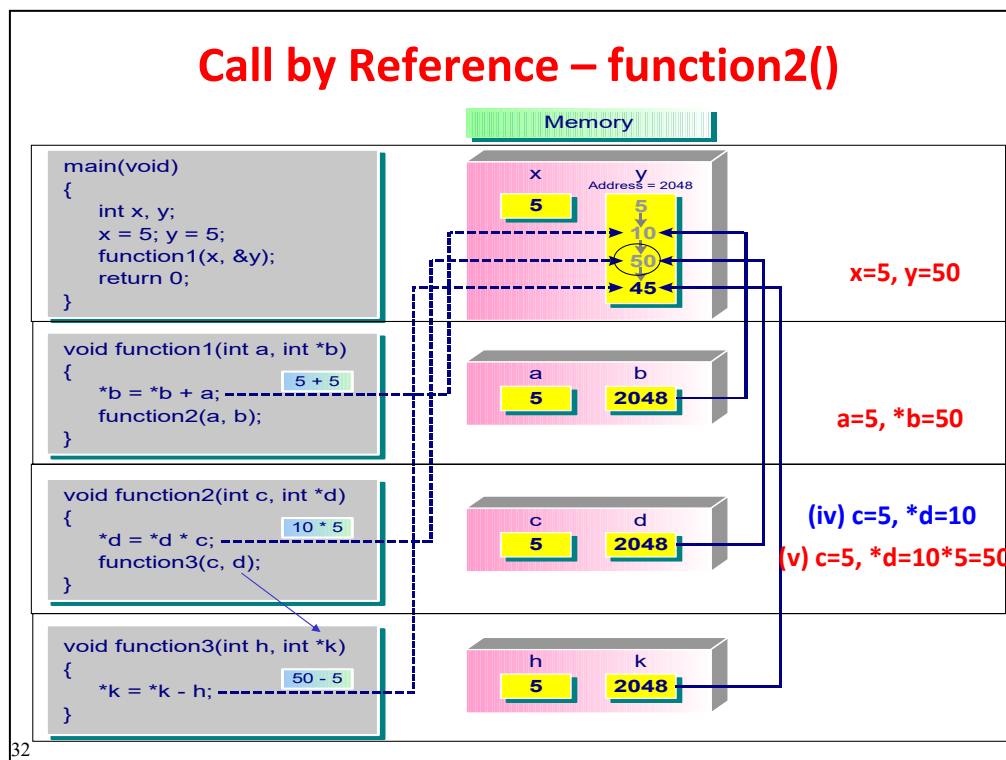
Call by Reference: main()

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** respectively.
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.



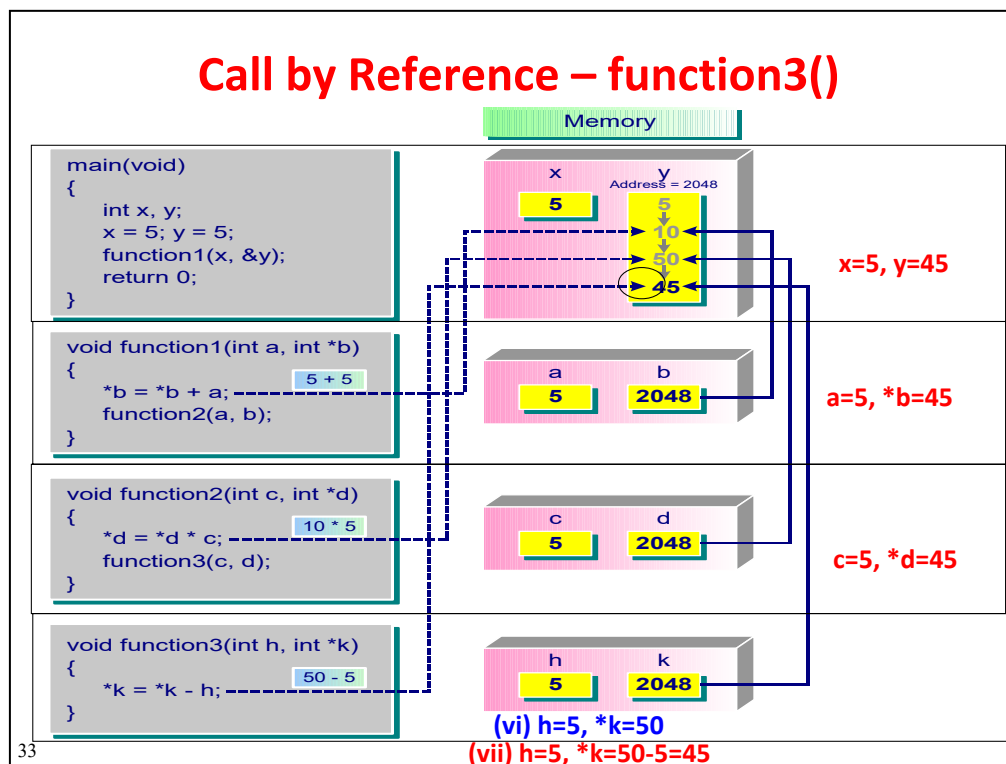
Call by Reference: function1()

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: function2()

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: function3()

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 to **function2()** for execution and then terminates, which in turn returns to **function1()**, and then terminates and finally returns to the **main()** function.

Call by Reference – Example 2

	x	y	a	*b	c	*d	h	*k	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

34

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 does not change 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) First, when we need to pass more than one value back from a function.
 - b) Second, 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 `int ***ppp;` etc. The idea remains the same.

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 **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=2;
*p=2;
**p=2;

```



Thank You!

37

Thank You

1. Thanks for watching the lecture video.