4
Pointers

1

Review

- What have you learnt so far:
  - Basic Sequential Programming (data, operators, simple I/O)
  - Branching (if-else, switch, conditional operator)
  - Looping (for, while, do…while)
  - Functions
- With Functions, you may write better structured programs. However, in functions, you can only **return ONE** value to the calling function. If we need to return more than one value (such as passing x,y coordinates of a point to the calling function), we could:
  - Use **global variables**: however, it is not recommended as the code will not be well structured ….

2

Review

- Instead, we may achieve that through **Pointers**, you may use **call by reference** via pointers to pass more than one value to the calling function…..
- Using pointers to **return values** to the calling function is highly recommended instead of using global variable. By doing so, each function will be self-contained and it is much easier to debug any error by localizing it within the function.

3

This Lecture

- At the end of this lecture, you will be able to understand the following:
  - Address Operator
  - Pointer Variables
  - Call by Reference

4

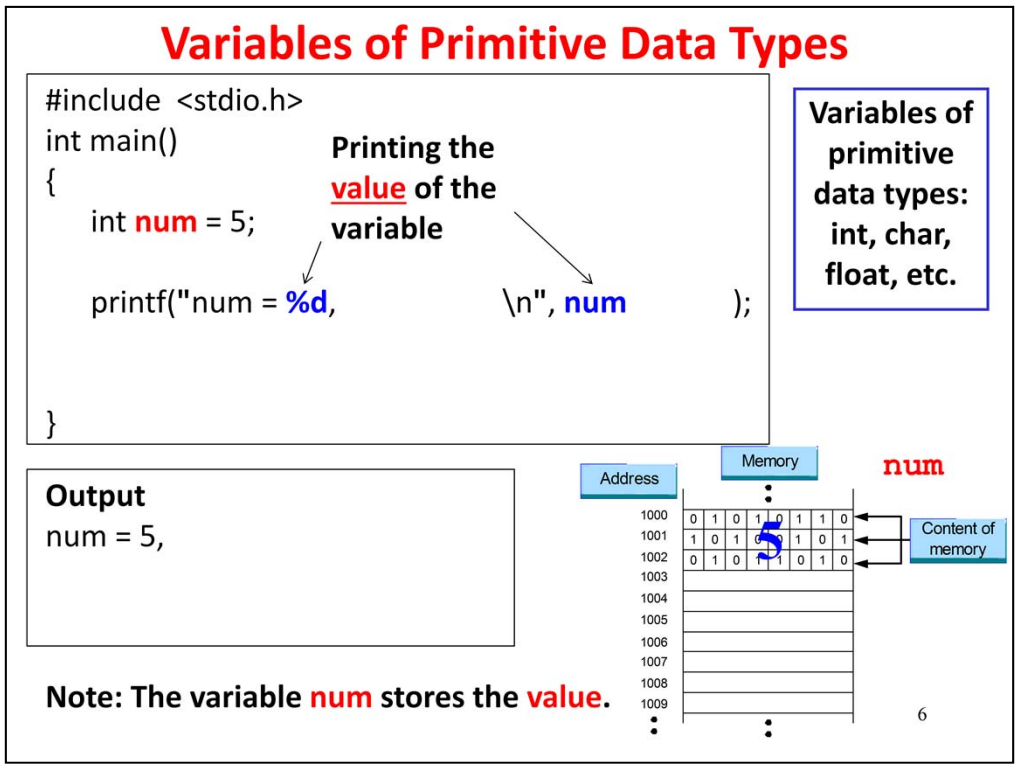
Pointers

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. 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. In this lecture, we discuss the concepts of pointers including address operator, pointer variables and call by reference.

Pointers

- **Address Operator**
- Pointer Variables
- Call by Reference

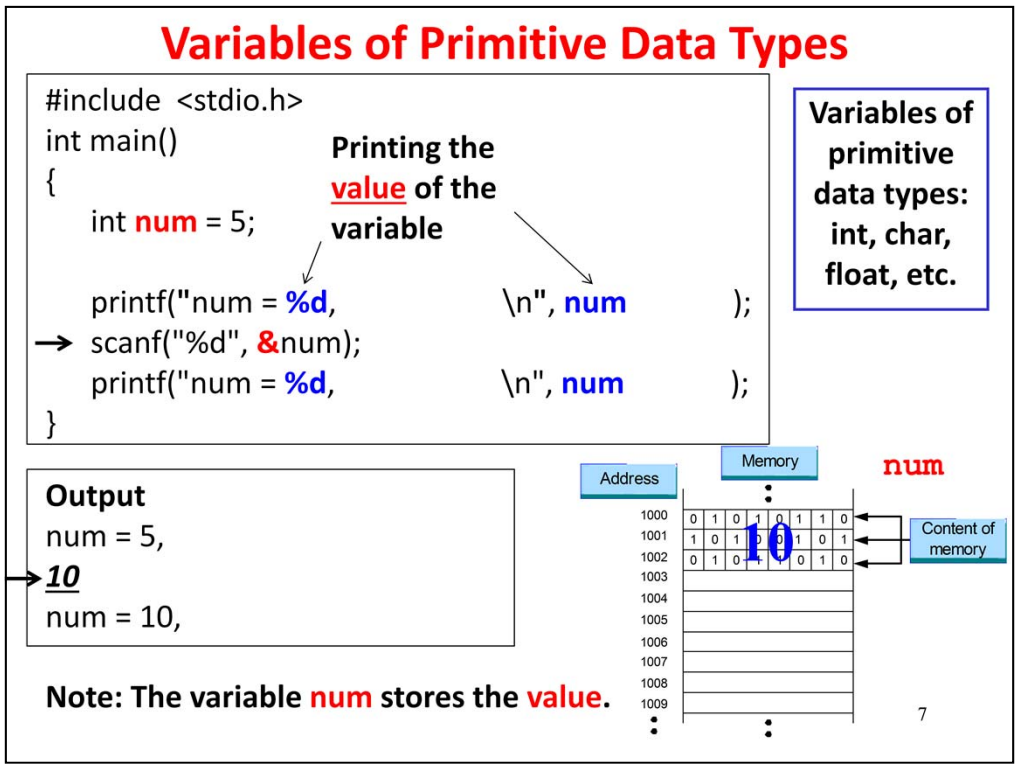
5



Variables of Primitive Data Types

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. 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.

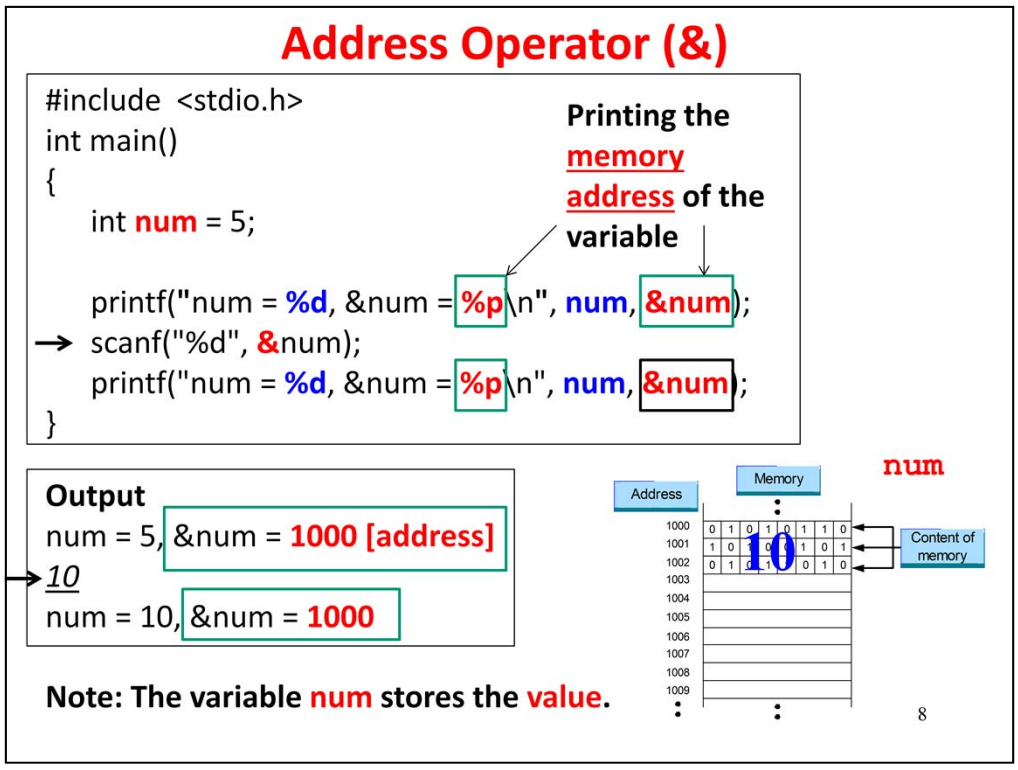
Note that variables of primitive data types such as **int**, **char**, **float**, **double**, etc. are used to store the actual data. For example, in the program, the variable **num** is declared as an **int**. When the variable **num** is initialized with the value 5, the memory location of the variable 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

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. 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.

Note that variables of primitive data types such as **int**, **char**, **float**, **double**, etc. are used to store the actual data. 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.



Address Operator

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);

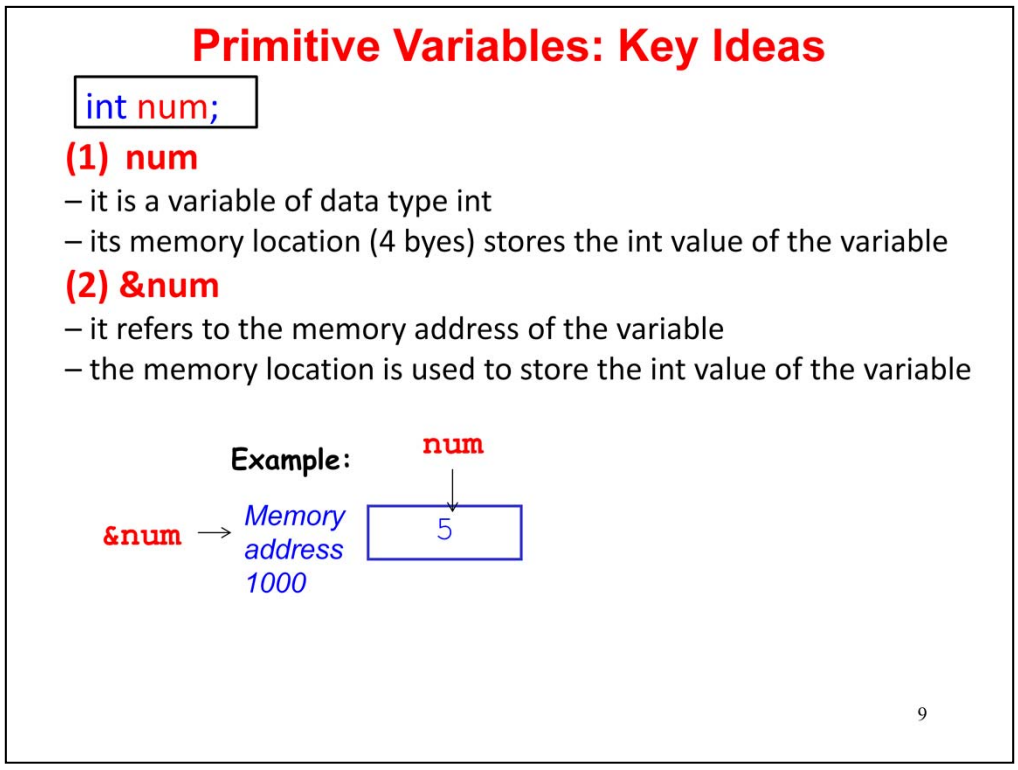
In the **printf()** statement, it prints two values on the screen. The first one is the value 5 that is the initialized value stored at the memory location of the variable **num**. The other is the memory address at which the value 5 is stored. The address of this memory location is 1024. However, as 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.

The statement

scanf("%d", &num);

reads in a value of 10 from the standard input, and then stores the value into the address location of the variable **num**.

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

There are two important ideas related to variables:

1. Primitive variable (**num**) – it stores an variable value of data type int.
2. Address of Variable (**&num**) – It refers to the memory address of the variable. The memory location is used to store the variable value.

Pointers

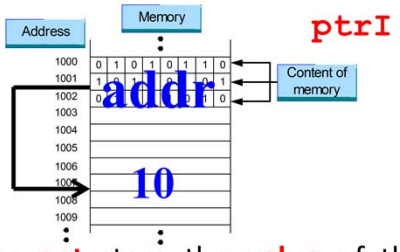
- Address Operator
- **Pointer Variables**
- Call by Reference

10

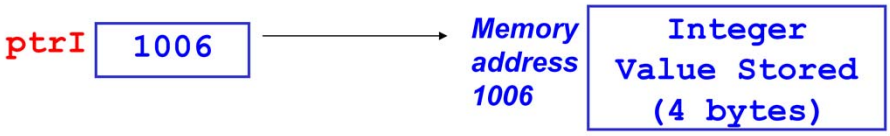
Pointer Variables: Declaration

- **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,



11

Pointer Variables: Declaration

Apart 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. 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.

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**.

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 (*), e.g. ***ptrI**.

Pointer Variables: Declaration (Cont'd.)

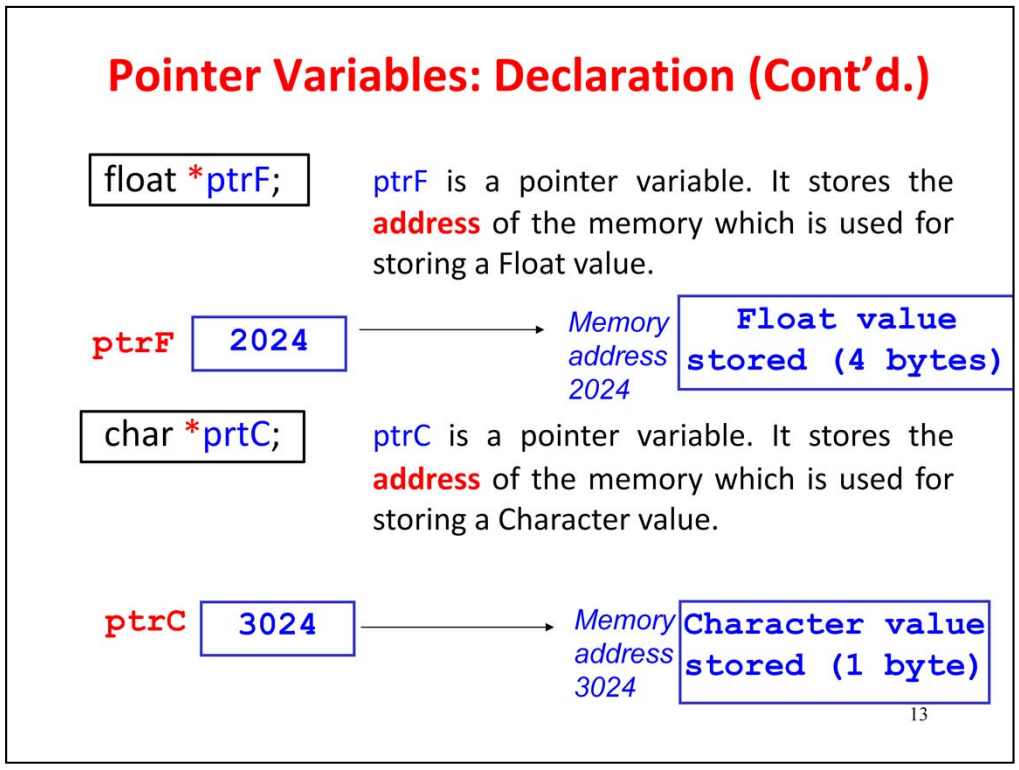
- Analogy:
  (1) Address on envelope → your home

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

12

Pointer Variables: Delcaration

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. It is also similar to a bank account, which contains the saving information and the bank location that stores the real saving money. The saving money can be referred to via the bank account.



Pointer Variables: Delcaration

The statement

float *ptrF;

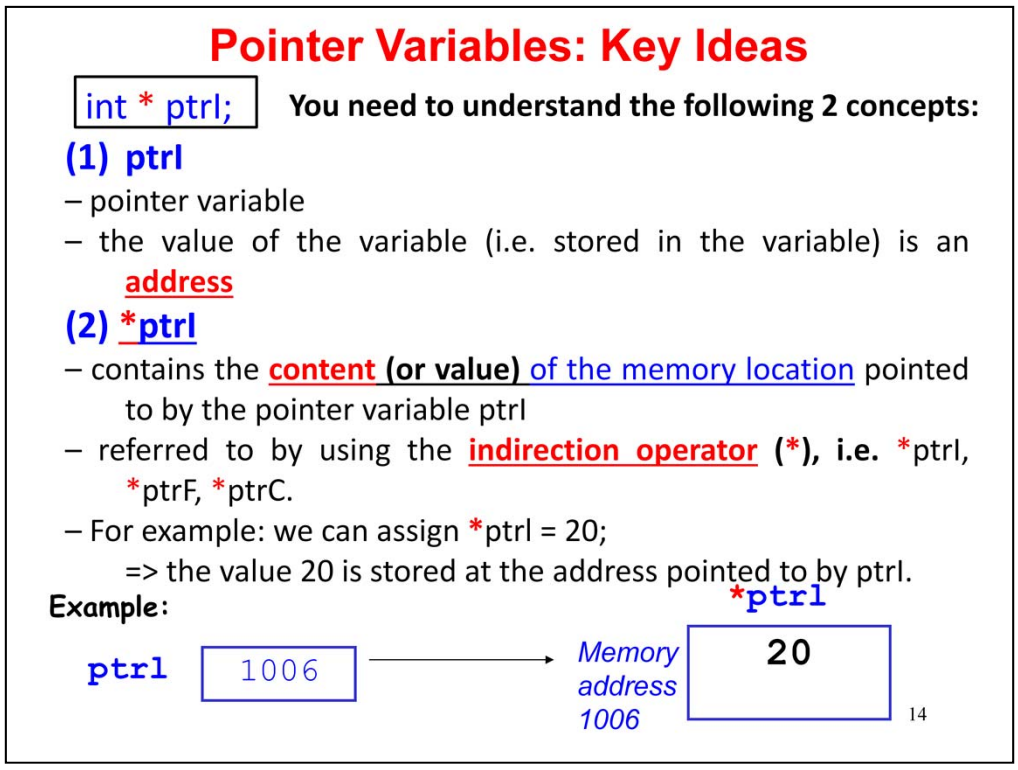
declares a pointer variable **ptrF** that points to the address of a memory location that is used to store a **float**.

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**.

When a pointer is declared without initialization, memory is allocated to the pointer variable. However, no data or address is stored there.



Pointer Variables: Key Ideas

There are two important concepts related to pointers:

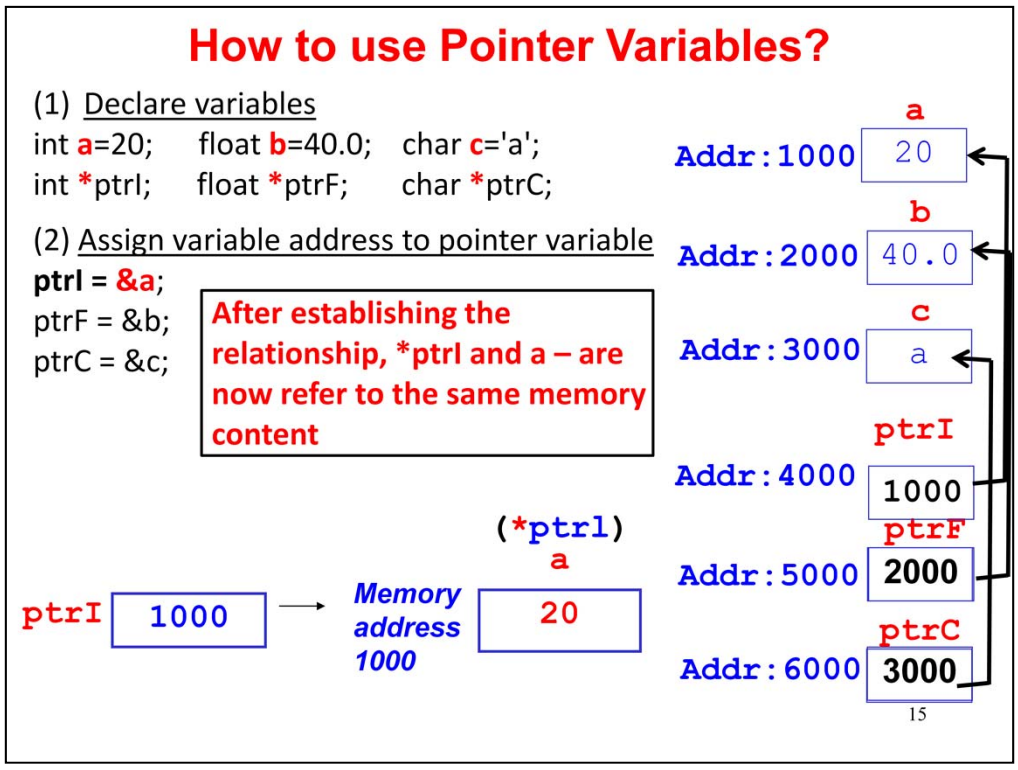
1. Pointer variable (**ptr**) – it stores an address which refers to the location that stores the actual data.
2. Indirection operator (***ptr**) – Using the indirection operator (*), e.g. ***ptr**, we can then retrieve the actual value pointed to by the pointer variable.

For example, if we declare a pointer variable as follows:

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

The statement will update the memory location pointed to by (contained in) the pointer variable to 20.

Then, 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.



Assigning Variable Address to Pointer Variable

The value of a pointer is an address. After the declaration statements

int a=20; float b=40.0; char c='a';

int *ptrI; float *ptrF; char *ptrC;

the variables **a**, **b** and **c**, and pointer variables **ptrI**, **ptrF** and **ptrC** are created. We can then assignment variable address to pointer variable as follows:

ptrI=&a;

ptrF=&b;

ptrC=&c;

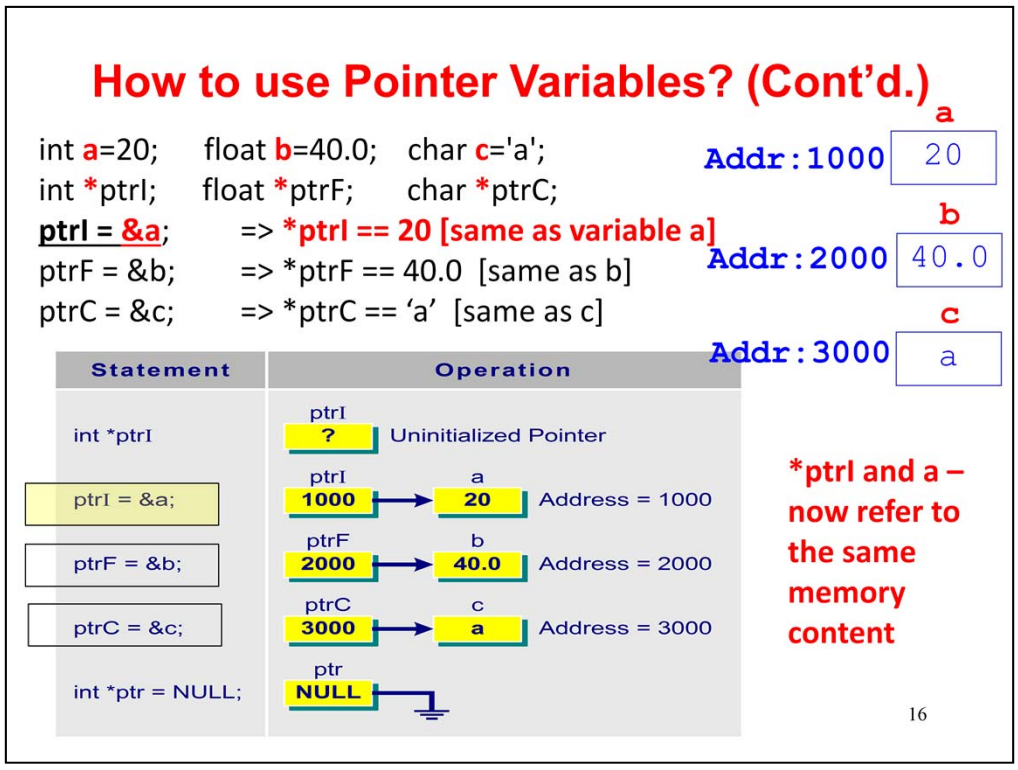
The value of a pointer variable is an address. The pointer variables then point to the memory locations that are used to store the values. The statement

ptrI = &a;

is used to assign the address of the memory location of **a** to the pointer variable **ptrI**. It is similar to the other assignment statements.

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;



Assigning Variable Address to Pointer Variable

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**.

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**.

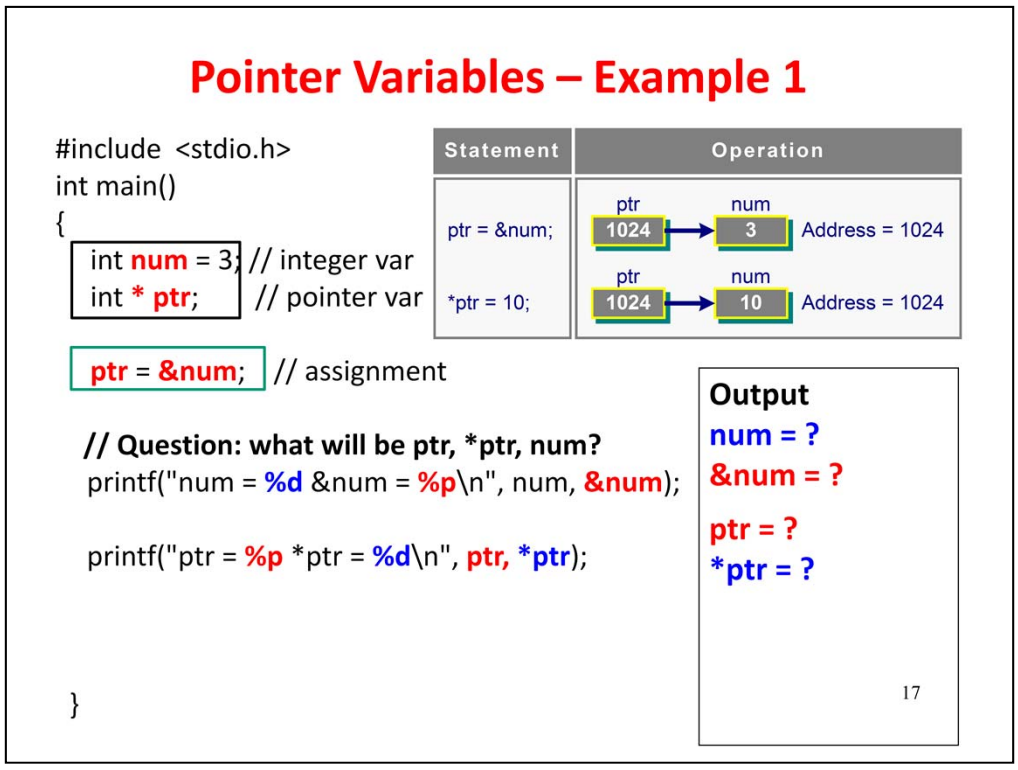
It implies that:

- ***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'.

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

In the program, a pointer variable **ptr** is declared to point to a variable of type **int**. The statement

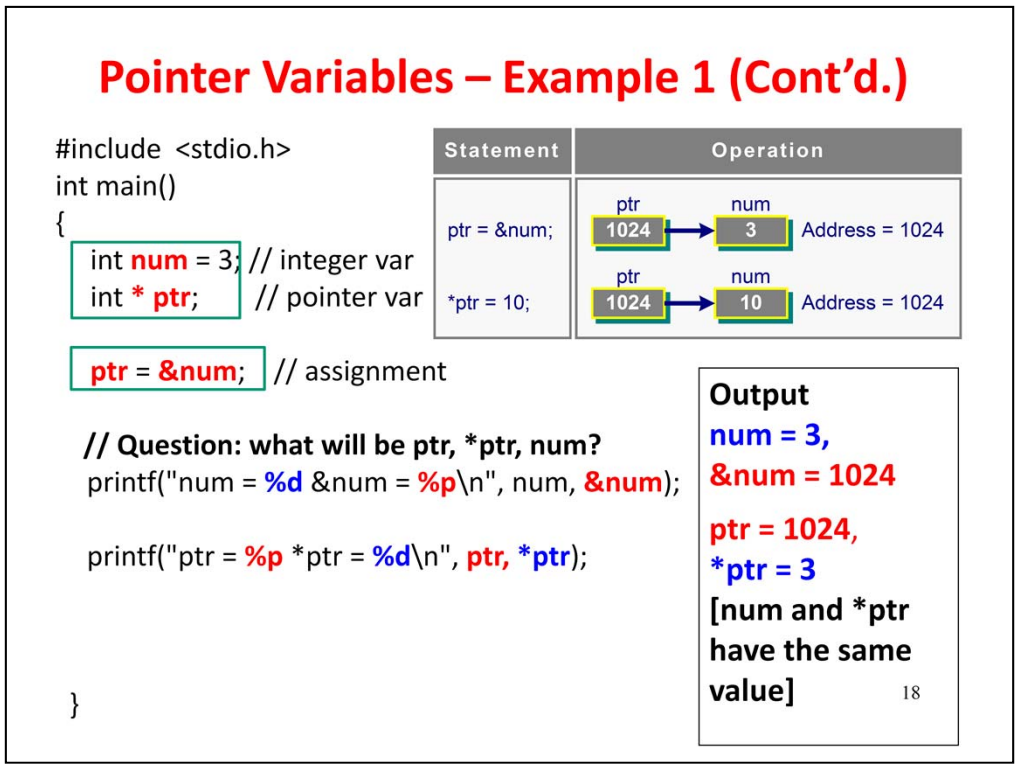
ptr = #

assigns the address of the variable **num** to the pointer variable **ptr**.

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

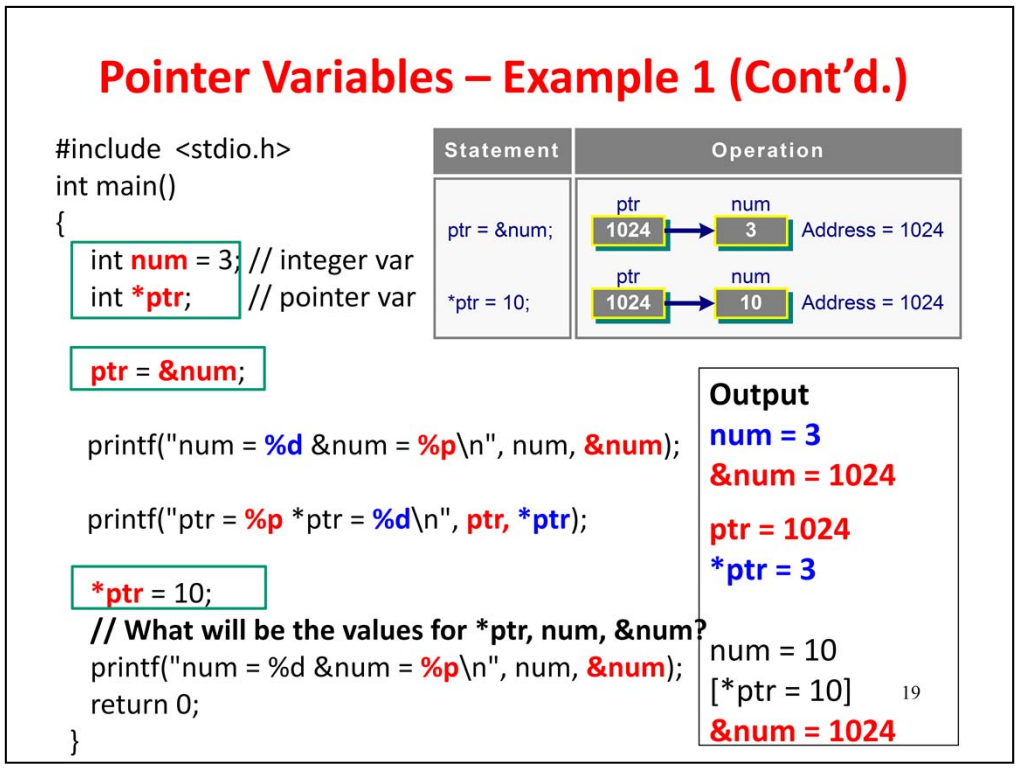
The primitive type variable **num** stores the value of 3, and the address of the memory location of the variable **num** is 1024.

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.

After the assignment statement:

ptr = #

the pointer variable **ptr** stores the address of the memory location of the variable **num**. 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).

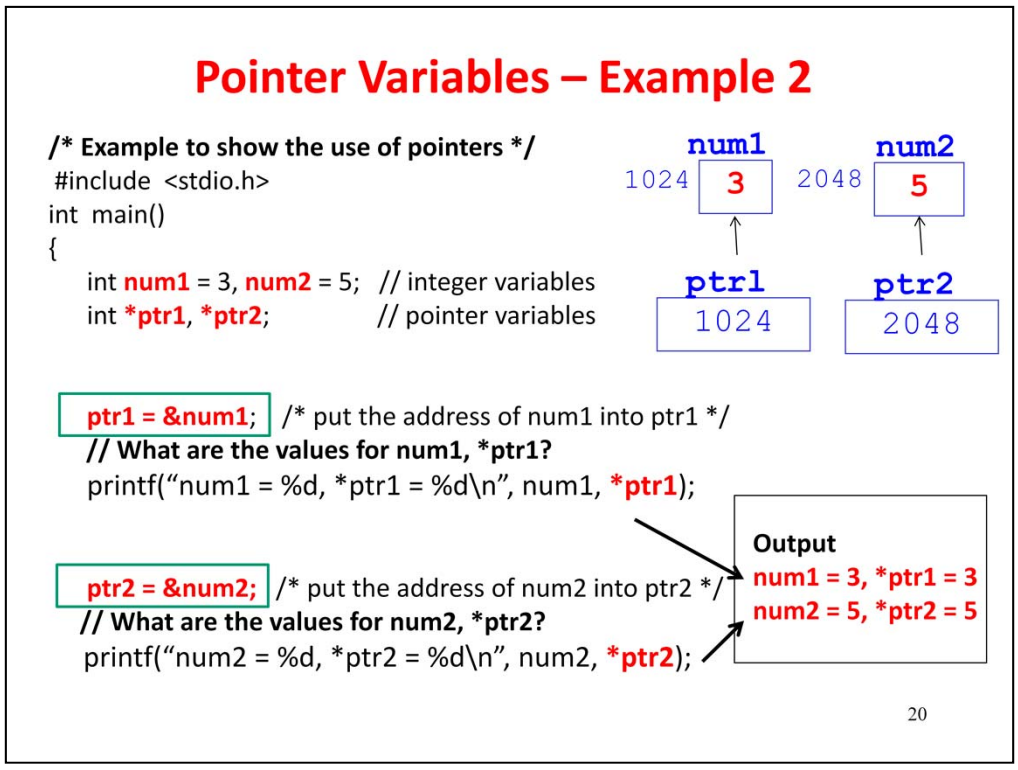


Pointer Variables: Example 1

The statement

***ptr = 10;**

assigns the value 10 to the variable **num**. 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**. 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

In the program, the statements

ptr1 = &num1;

ptr2 = &num2;

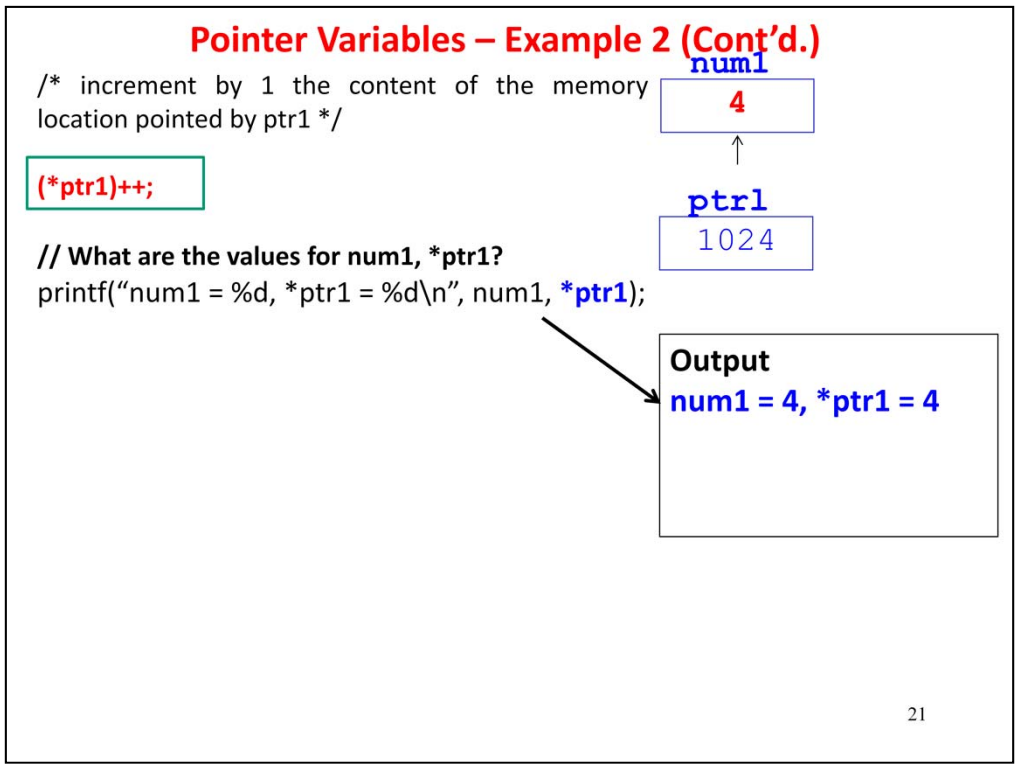
assign the address of **num1** into **ptr1**, and the address of **num2** into **ptr2**.

Therefore, we have

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

and

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



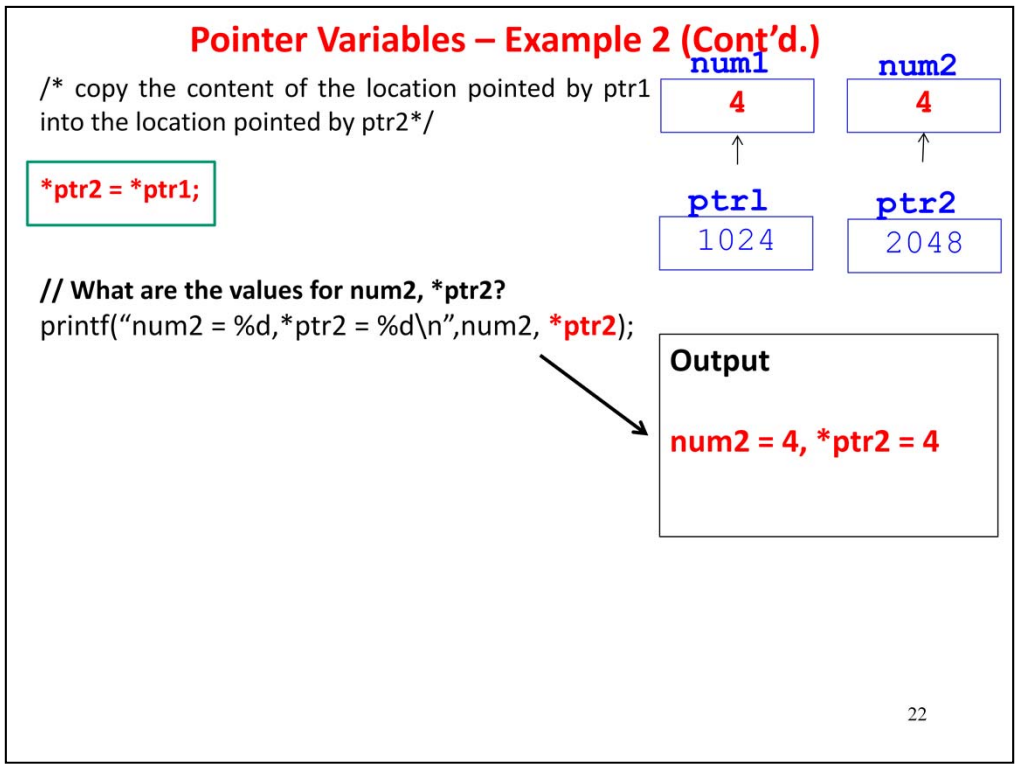
Pointer Variables: Example 2

The statement

(*ptr1)++;

increments 1 to the content of the memory location pointed to by **ptr1**.

Therefore, we have ***ptr1**= 4, and **num1** =4.



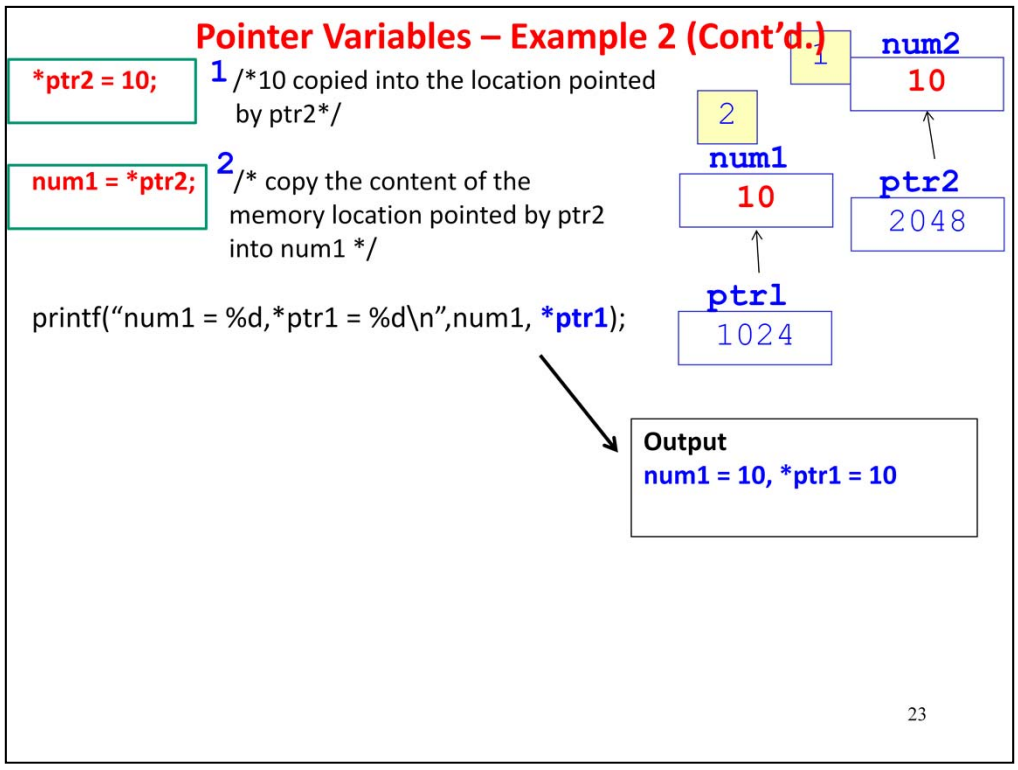
Pointer Variables: Example 2

The statement

***ptr2 = *ptr1;**

copies the content of the location pointed to by **ptr1** into the location pointed to by **ptr2**.

Since ***ptr1** = 4, we have ***ptr2** = 4 and **num2** = 4.



Pointer Variables: Example 2

The statement

***ptr2** = 10**;**

assigns the value 10 to the content of the memory location pointed to by **ptr2**.

Therefore,

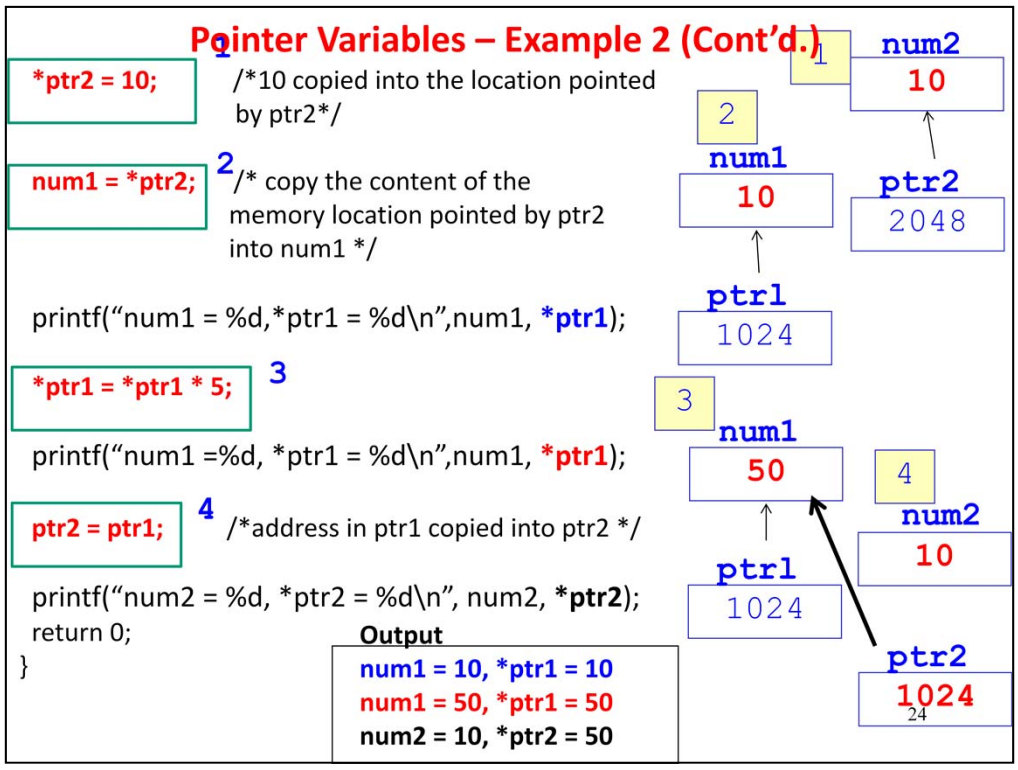
***ptr2** = 10, and **num2** = 10.

The statement

num1 = *ptr2;

copies the content of the memory location pointed to by **ptr2** into **num1**.

Since ***ptr2** = 10, **num1** = 10, we have **num1** = 10, and ***ptr1** = 10.



In the statement

***ptr1 = *ptr1 * 5;**

since ***ptr1** = 10, we have ***ptr1***5 = 50. 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.

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. However, the value of **num2** is not changed, we have **num2** = 10.

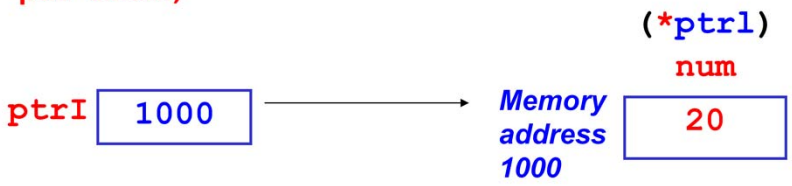
The concepts of using pointers in the program are summarized as follows:

- **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: Key Ideas

1. Declare variables and pointer variables:
   int num=20;
   int *ptrI;

2. Assign the address of variable to pointer variable:
   ptrI=#



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

25

Call by Reference: Key Ideas

There are two key ideas on using pointer variables:

1. Declare variables and pointer variables:
   int num=20;
   int *ptrI;

2. Assign the address of variable to pointer variable:

   ptrI=#

Self-Practice Exercise

26

Problem: Pointer Variables

Assume the following declaration:

```
int number;
int *p;
```

Assume also that the address of number is 7700 and the address of p is 3478.

3478		p
	. . .	
7700		number

For each case below, determine the value of

(a) number (b) &number (c) p (d) &p (e) *p

All of the results are cumulative.

(i) p = 100; number = 8
(ii) number = p
(iii) p = &number
(iv) *p = 10
(v) number = &p
(vi) p = &p

Check Your Answer at the end of the chapter

27

	a	b	c	d	e
I	8	7700	100	3478	?
II	100	7700	100	3478	?
III	100	7700	7700	3478	100
IV	10	7700	7700	3478	10
V	3478	7700	7700	3478	3478
VI	3478	7700	3478	3478	3478

tricky!

Pointers

- Address Operator
- Pointer Variables
- **Call by Reference**

28

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 ***.

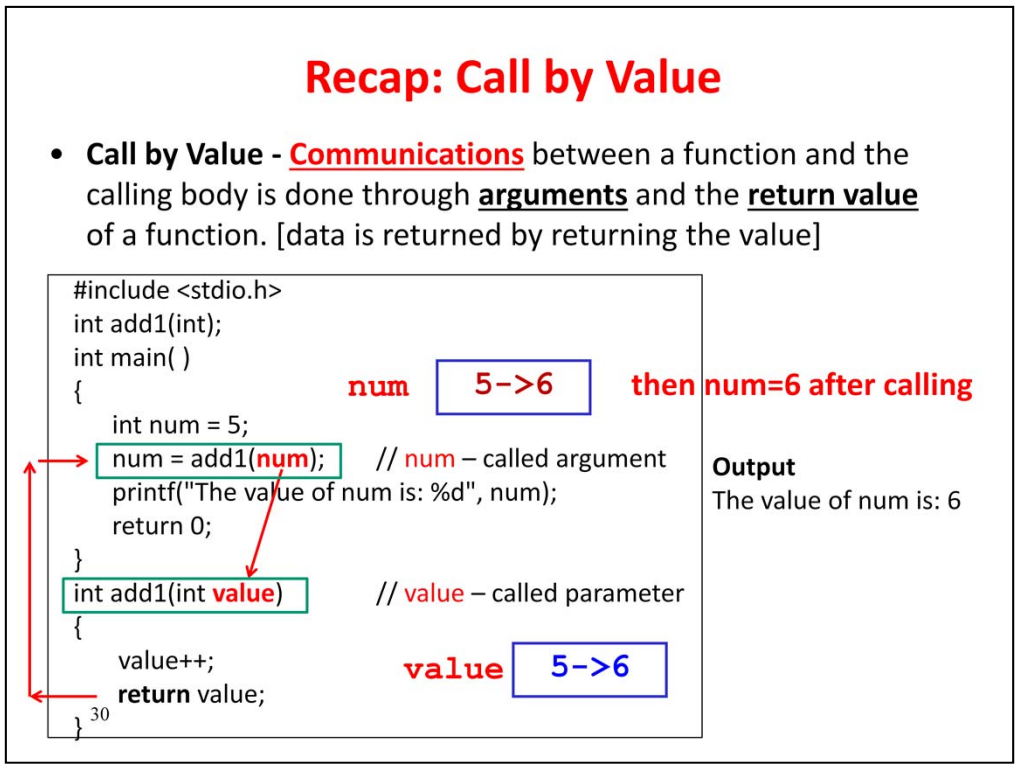
```
E.g. void distance(double *x, double *y)
```

29

Call by Reference

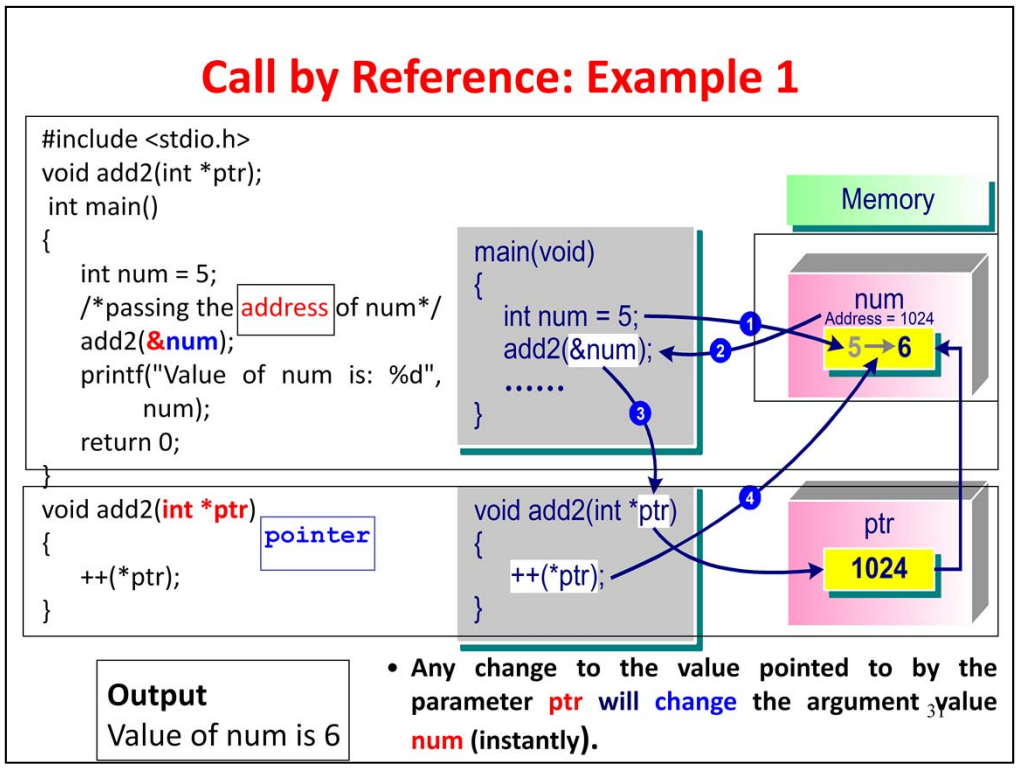
Parameter passing between functions can be done either through call by value or call by reference. Call by value has been discussed in the last lecture on functions.

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. In a function call, the **arguments** must be **pointers** (or using address operator as the prefix). In the parameter declaration list, the **parameters** must be prefixed by the **indirection operator**.



Recap: Call by Value

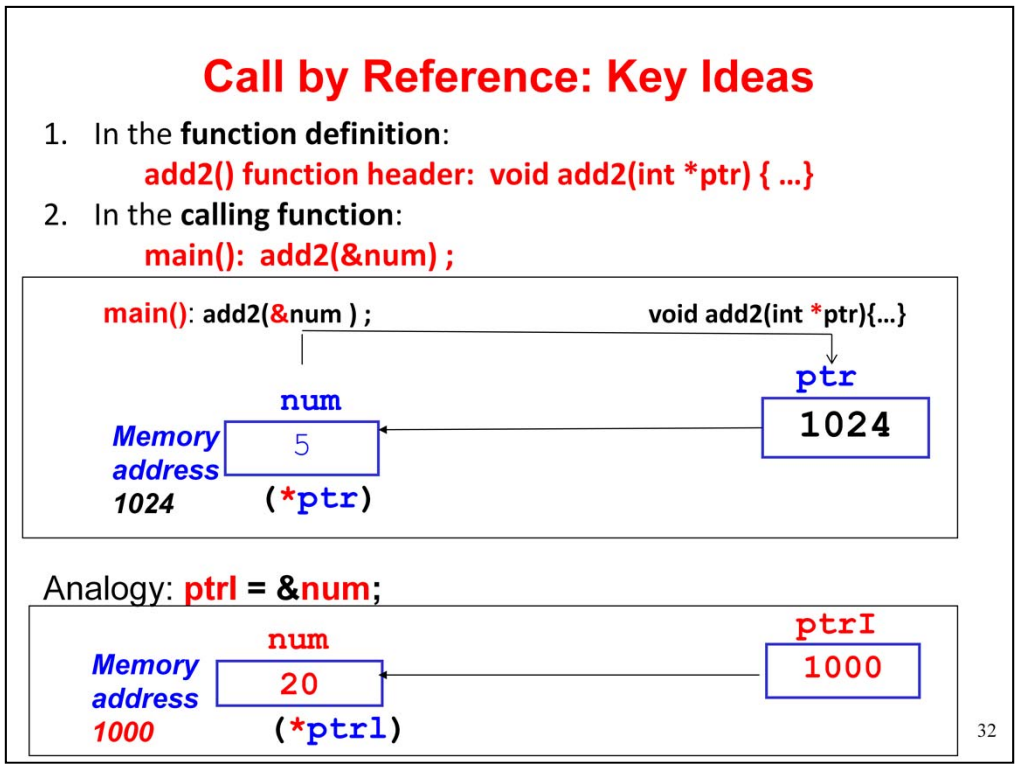
This example was used in the chapter on Functions.



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.

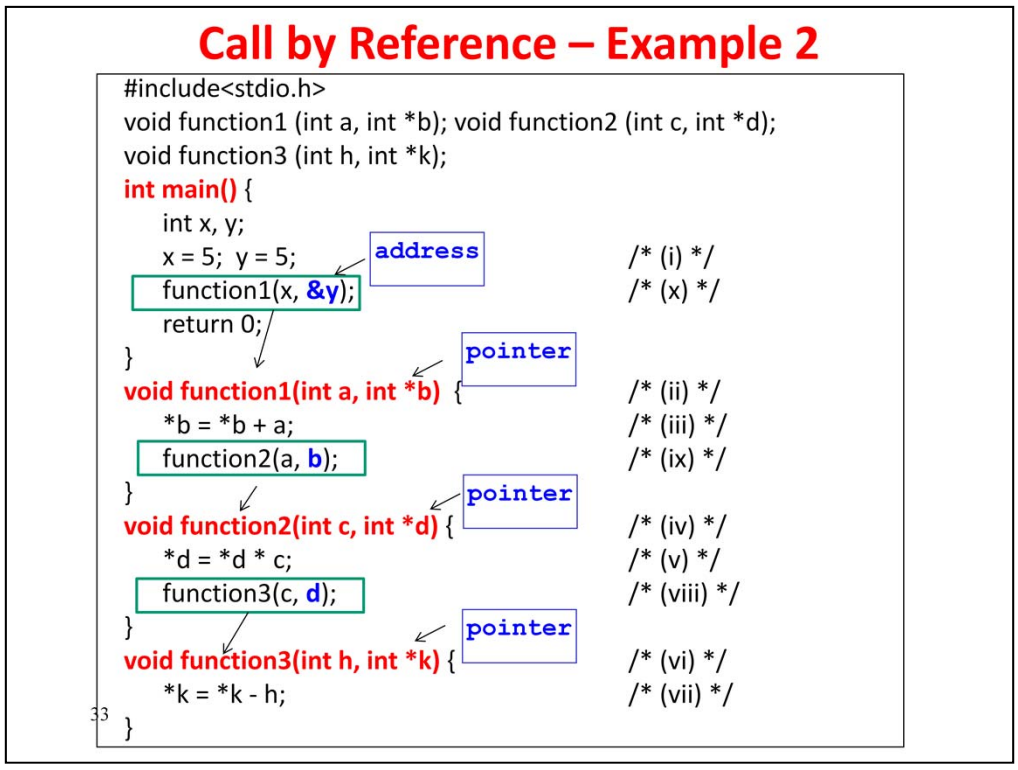
In this example, note that the parameter variable **ptr** in **add2()** is used to store the address of the variable **num** in **main()**. After passing the variable address of **num** into the parameter variable **ptr**, all the operations on **ptr** in the function **add2()** will update the content of the variable **num** indirectly.



Call by Reference: Key Ideas

There are two key ideas related to call by reference:

1. In the function, the parameter must be prefixed by the indirection operator: **void add2(int *ptr);**
2. In the calling function (e.g. **main()**), the arguments must be pointers (or using address operator as the prefix): **add2(&num);**



Call by Reference: Example 2

In the function definitions:

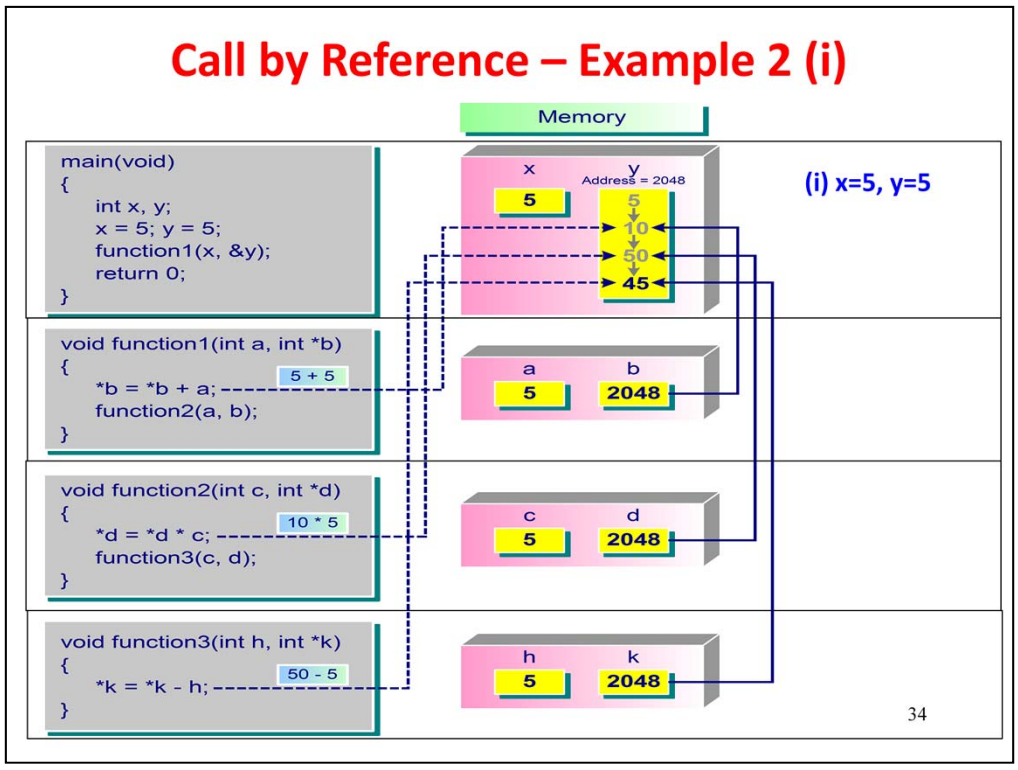
void function1(int a, int *b)

void function2(int c, int *d)

void function3(int h, int *k)

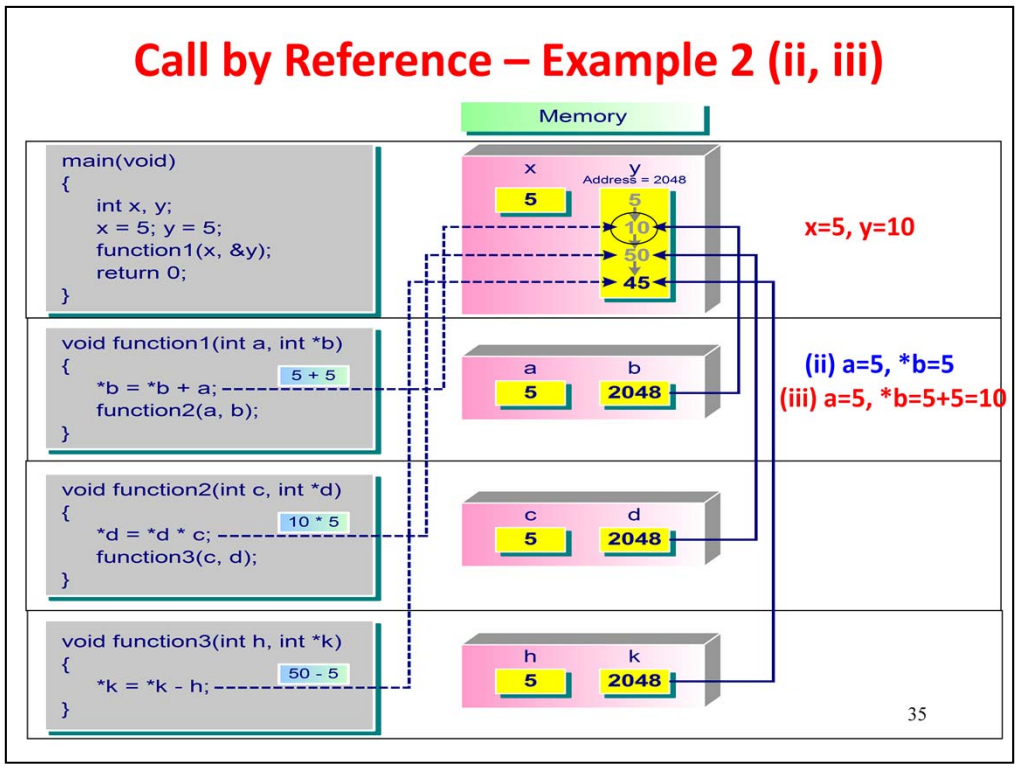
the parameters **a**, **c** and **h** are passed into the functions using call by value, whereas the parameters **b**, **d** and **k** are passed into the functions using call by reference, i.e. addresses are passed to the functions instead of actual values.

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

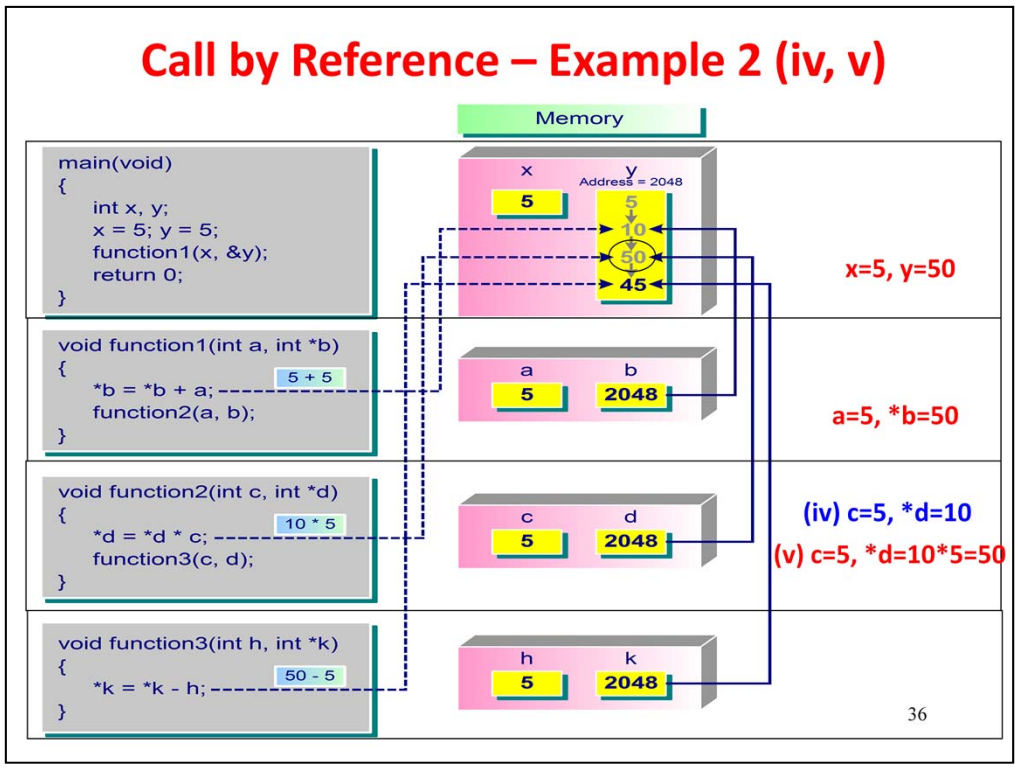
When the program starts execution, in the **main()** function, memory locations are allocated to the variables **x** and **y** accordingly. The two variables are assigned with the value of 5. Therefore, the memory locations of the variables **x** and **y** store the value of 5 directly. The **main()** 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**. 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

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**. 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.

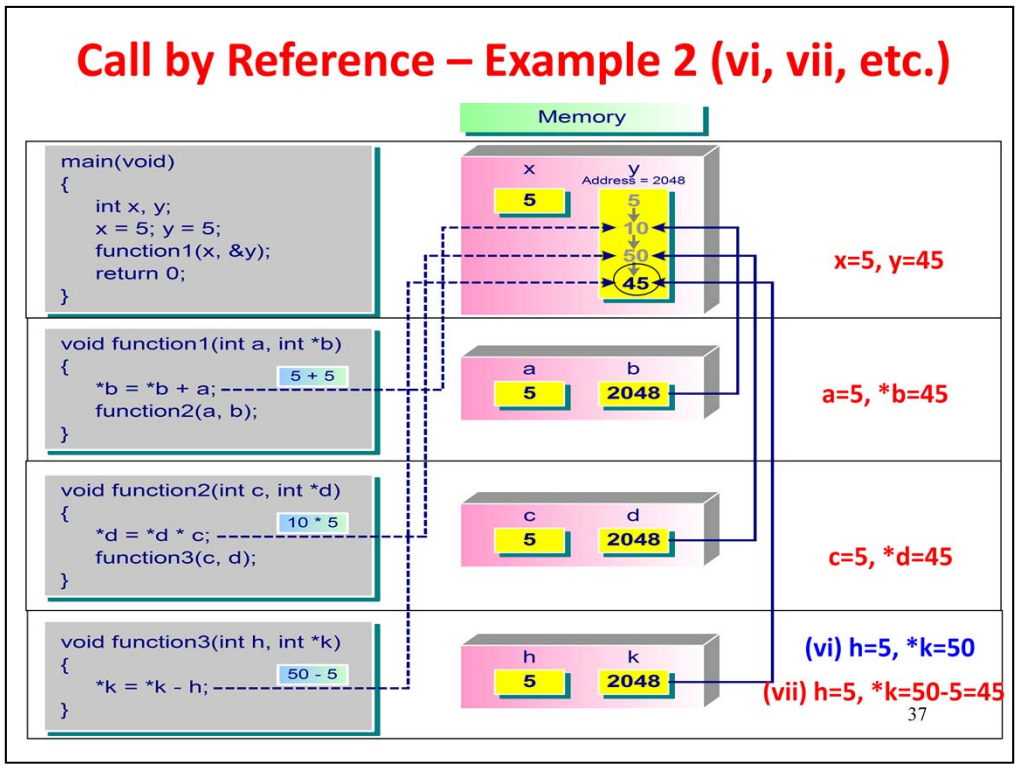
After that, **function1()** calls **function2()** by passing in the values of **a** and **b** into the parameters **c** and **d** respectively.



Call by Reference: Example 2

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**. 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.

After that, **function2()** calls **function3()** by passing in the values of **c** and **d**.



Call by Reference: Example 2

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**. 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.

Returning Control

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.

Call by Reference – Example 2

	x	y	a	*b	c	*d	h	*k	remarks
(i)	5	5	-	-	-	-	-	-	
(ii)	5	5	5	5	-	-	-	-	
(iii)	5	10	5	10	-	-	-	-	
(iv)	5	10	5	10	5	10	-	-	
(v)	5	50	5	50	5	50	-	-	
(vi)	5	50	5	50	5	50	5	50	
(vii)	5	45	5	45	5	45	5	45	
(viii)	5	45	5	45	5	45	-	-	
(ix)	5	45	5	45	-	-	-	-	
(x)	5	45	-	-	-	-	-	-	

38

Call by Reference: Example 2

The values for each variable and parameter are summarized in the table.

Self-Practice Exercise

39

Problem: Call by Reference

- What will be the output of the following program?

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);    /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);    /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);    /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);    /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);    /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);    /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);    /* line (vii) */
}
void function2( int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
}
```

Check Your Answer at the end of the chapter

40

	h	k
(i)	5	15
(ii)	5	15
(iii)	5	5
(iv)	200	200
(v)	-100	-100
(vi)	5	15
(vii)	100	100
(viii)	5	15
(ix)	200	200

When to Use Call by Reference

When to use call by reference:

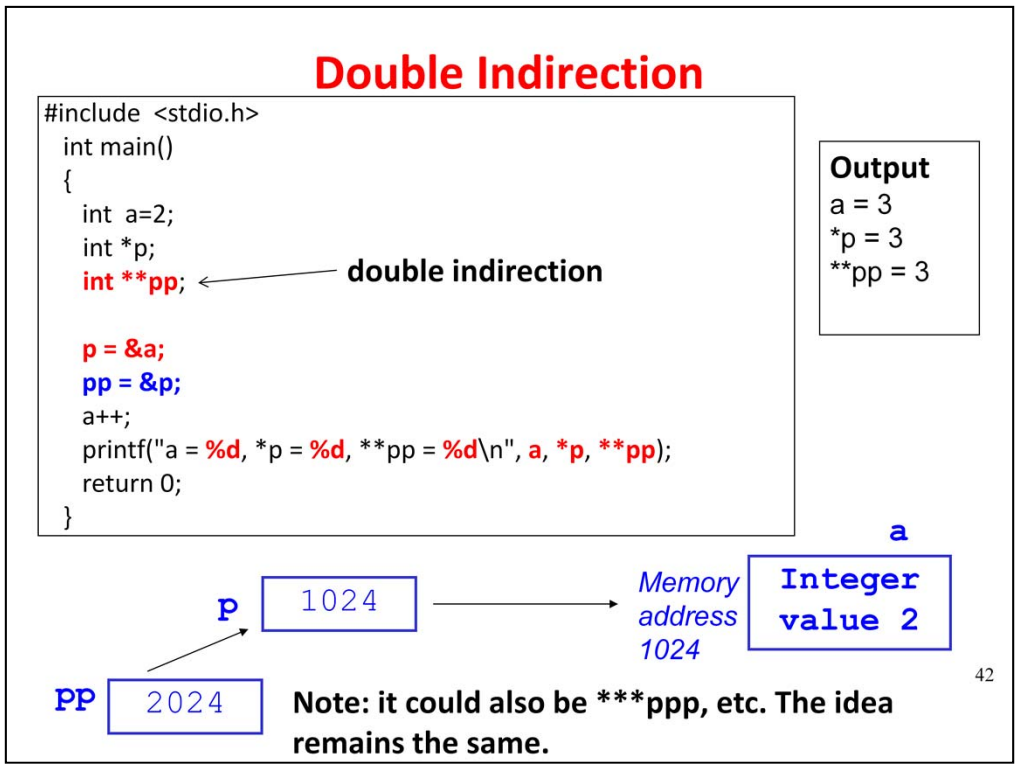
(1) When you need to pass **more than one value** back from a function.

(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.

41

When to Use Call by Reference

Generally, call by reference is used when we need to pass more than one value back from a function, or 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

We have seen examples on using indirection operator. Double indirection is also quite commonly used in C programming.

A variable can be declared as

int **pp;

using double indirection.

In this case, the variable **pp** will be used to store an address of another pointer variable (e.g. **p**) (i.e. it points to another pointer variable), which will in turn store the address of a variable (e.g. **a**) of the corresponding primitive type.

Therefore, after the variable declaration and assignment, we will have the output:

a=3;

***p=3;**

****p=3;**

Programming Problem

43

Question: How to write a function to swap the contents of 2 variables?

- The following program shows how to write the main() function to swap the contents of two variables. It is easy to do so in main(). Could the swapping operation be implemented as a function.

Inside the main() function:

```
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    int temp;

    printf("Before: a = %d, b = %d\n", a, b);
    temp = a;      a =b;      b = temp;
    printf("After: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output

```
Before: a = 5, b = 10

After: a = 10, b = 5
```

44

Programming Problem

The purpose of the program is to illustrate the use of call by reference. The problem is to write a function that can swap the values of two parameters of the function. As shown in this program, it is quite straightforward to implement the swapping operation inside the **main()** function.

Question: How do you implement a function called **swap()** to swap the values of 2 variables?

Problem: Swapping the Values of Two Variables with a Function

- Swapping the contents of 2 variables through a function.

```
#include<stdio.h>
void swap(int *, int *);
int main() {
    int a = 5, b = 10;

    printf("Before: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After: a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int *x, int *y) {
    /* Write your code here */
 }
```

Question: How to implement a function for the swapping operation?

Using Call by Reference through Pointers

45

Programming Problem: Implement the Function swap()

In the **main()** function of the program template, it declares two variables **a** and **b**. When the function **swap()** is called, the addresses of the two variables are passed into the calling function. In the function **swap()**, it swaps the contents of the two parameters **x** and **y**, which contain the addresses of the variables **a** and **b** passed in from the **main()** function. It is done via **call by reference**, as the function is required to return two values back to the calling function.

Question: How do you implement the function **swap()**?

A Function that Swaps Two Variables: Suggested Code

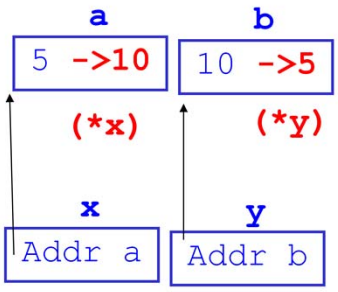
```
/* Writing a function to swap two values.
Author: S.C. Hui; */
#include<stdio.h>
void swap(int *, int *);
int main() {
    int a = 5, b = 10;

    printf("Before: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After: a = %d, b = %d\n", a, b);
    return 0;
}
```

Output
Before: a = 5, b = 10
After: a = 10, b = 5

Passing Address

```
void swap(int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```



46

Programming Problem: Suggested Code

In the **swap()** function, a simple swap operation is performed based on the parameters **x** and **y** via call by reference. Note that we use **pointer dereferencing** for the swapping operation:

```
temp = *x;
*x = *y;
*y = temp;
```

A temporary variable **temp** is used to store the temporary data during the swapping operation.

Check Your Answers

47

int number;
int *p;

Problem: Pointer Variables

	Mem addr	Memory content	var	(a) num	(b) &num	(c) p	(d) &p	(e) *p
(i) p=100; number=8	3478	100	p	8	7700	100	3478	Content of mem location 100
	7700	8	number					
(ii) number=p	3478	100	p	100	7700	100	3478	Content of mem location 100
	7700	100	number					
(iii) p=&number	3478	7700	p	100	7700	7700	3478	100
	7700	100	number					
(iv) *p=10	3478	7700	p	10	7700	7700	3478	10
	7700	10	number					
(v) number = &p	3478	7700	p	3478	7700	7700	3478	3478
	7700	3478	number					
(vi) p=&p	3478	3478	p	3478	7700	3478	3478	3478
	7700	3478	number					

48

Problem: Pointer Variables

i. (a) number is 8 (b) &number is 7700 (c) p is 100 (d) &p is 3478 (e) *p is the content of the memory location 100.

ii. (a) number is 100 (b) &number is 7700 (c) p is 100 (d) &p is 3478 (e) *p is the content of the memory location 100.

iii. (a) number is 100 (b) &number is 7700 (c) p is 7700 (d) &p is 3478 (e) *p is 100.

iv. (a) number is 10 (b) &number is 7700 (c) p is 7700 (d) &p is 3478 (e) *p is 10.

v. (a) number is 3478 (b) &number is 7700 (c) p is 7700 (d) &p is 3478 (e) *p is 3478.

vi. (a) number is 3478 (b) &number is 7700 (c) p is 3478 (d) &p is 3478 (e) *p is 3478.

49

Problem: Call by Reference

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);    /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);    /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);    /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);    /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);    /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);    /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);    /* line (vii) */
}
void function2( int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
```

(1) h = 5, k = 15 line (i)

50

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);    /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);    /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);    /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);    /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);    /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);    /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);    /* line (vii) */
}
void function2(int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
```

(1) h = 5, k = 15 line (i)

(2) h = -100, k = -100 line (v)

51

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);   /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);   /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);   /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);   /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);   /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);   /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);   /* line (vii) */
}
void function2(int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
}
```

(1) h = 5, k = 15 line (i)

(3) h = 5, k = 15 line (ii)

(2) h = -100, k = -100 line (v)

52

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);    /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);    /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);    /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);    /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);    /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);    /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);    /* line (vii) */
}
void function2(int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
```

(1) h = 5, k = 15 line (i)

(3) h = 5, k = 15 line (ii)

(2) h = -100, k = -100 line (v)

(4) h = 5, k = 15 line (vi)

(5) h = 100, k = 100 line (vii)

53

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);   /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);   /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);   /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);   /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);   /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);   /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);   /* line (vii) */
}
void function2(int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
}
```

(1) h = 5, k = 15 line (i)

(3) h = 5, k = 15 line (ii)

(6) h = 5, k = 15 line (iii)

(2) h = -100, k = -100 line (v)

(4) h = 5, k = 15 line (vi)

(5) h = 100, k = 100 line (vii)

54

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);   /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);   /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);   /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);   /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);   /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);   /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);   /* line (vii) */
}
void function2(int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
```

(1) h = 5, k = 15 line (i)

(3) h = 5, k = 15 line (ii)

(6) h = 5, k = 15 line (iii)

(2) h = -100, k = -100 line (v)

(4) h = 5, k = 15 line (vi)

(5) h = 100, k = 100 line (vii)

(7) h = 5, k = 15 line (viii)

55

(8) h = 200, k = 200 line (ix)

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);    /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);    /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);    /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);    /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);    /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);    /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);    /* line (vii) */
}
void function2(int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
```

(1) h = 5, k = 15 line (i)

(3) h = 5, k = 15 line (ii)

(6) h = 5, k = 15 line (iii)

(9) h = 200, k = 200 line (iv)

(2) h = -100, k = -100 line (v)

(4) h = 5, k = 15 line (vi)

(5) h = 100, k = 100 line (vii)

(7) h = 5, k = 15 line (viii)

56

(8) h = 200, k = 200 line (ix)

Problem: Call by Reference

```
#include  <stdio.h>
void function0();
void function1(int h, int k);
void function2(int *h, int *k);
int main(){
   int h, k;
   h = 5;
   k = 15;
   printf("h = %d, k = %d\n", h, k);    /* line (i) */
   function0();
   printf("h = %d, k = %d\n", h, k);    /* line (ii) */
   function1(h, k);
   printf("h = %d, k = %d\n", h, k);    /* line (iii) */
   function2(&h, &k);
   printf("h = %d, k = %d\n", h, k);    /* line (iv) */
   return 0;
}
void function0(){
   int h, k;
   h = k = -100;
   printf("h = %d, k = %d\n", h, k);    /* line (v) */
}
void function1( int h, int k){
   printf("h = %d, k = %d\n", h, k);    /* line (vi) */
   h = k = 100;
   printf("h = %d, k = %d\n", h, k);    /* line (vii) */
}
void function2( int *h, int *k){
   printf("h = %d, k = %d\n", *h, *k); /* line (viii) */
   *h = *k = 200;
   printf("h = %d, k = %d\n", *h, *k); /* line (ix) */
}
```

The output:

h = 5, k = 15	line (i)
h = -100, k = -100	line (v)
h = 5, k = 15	line (ii)
h = 5, k = 15	line (vi)
h = 100, k = 100	line (vii)
h = 5, k = 15	line (iii)
h = 5, k = 15	line (viii)
h = 200, k = 200	line (ix)
h = 200, k = 200	line (iv)

57