

4

Pointers

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.

Pointers

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

Variables of Primitive Data Types

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

```
int main()
```

```
{
```

```
    int num = 5;
```

```
    printf("num = %d",
```

Printing the
value of the
variable

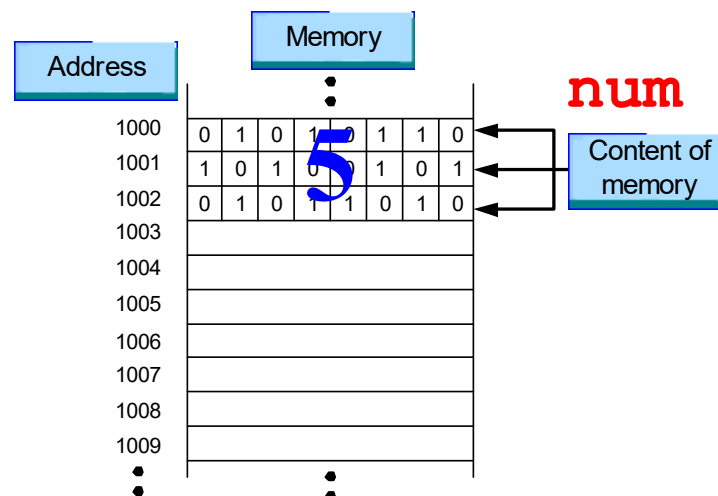
```
    \n", num );
```

```
}
```

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

Output

num = 5,



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

Variables of Primitive Data Types

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

```
int main()
```

```
{
```

```
    int num = 5;
```

```
    printf("num = %d",
```

```
→ scanf("%d", &num);
```

```
    printf("num = %d",
```

```
    }
```

Printing the
value of the
variable

\n", **num**);

\n", **num**);

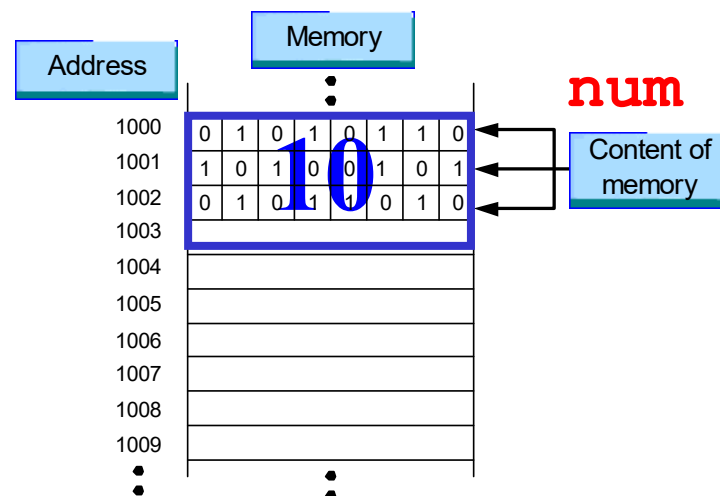
Variables of
primitive
data types:
int, char,
float, etc.

Output

num = 5,

→ 10

num = 10,



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

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);
```

```
}
```

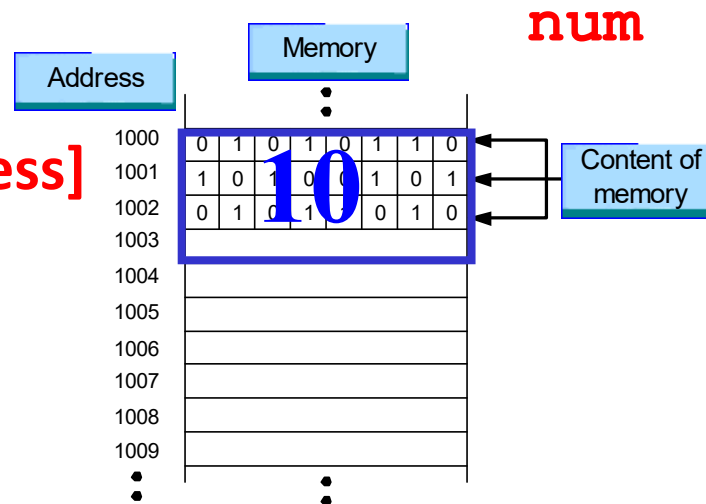
Printing the
memory
address of the
variable

Output

num = 5, &num = **1000 [address]**

→ 10

num = 10, &num = **1000**

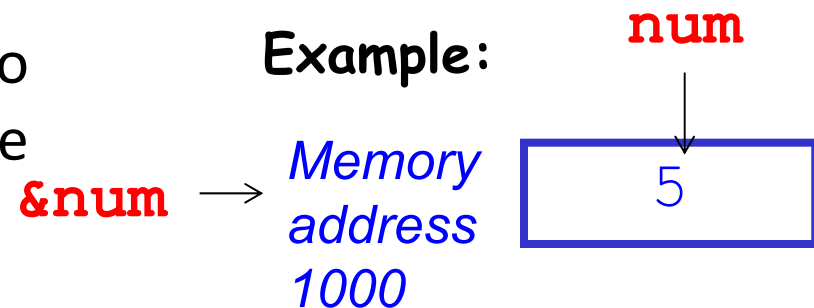


Primitive Variables: Key Ideas

```
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.

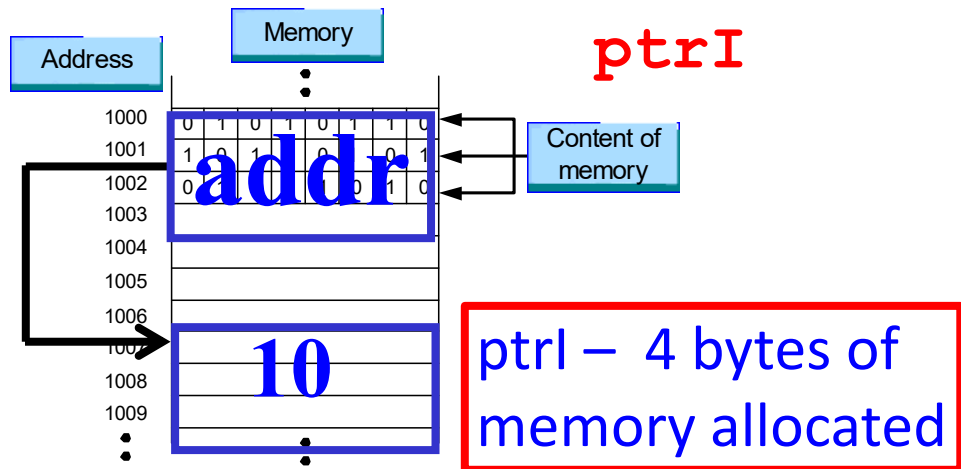
Pointers

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

Pointer Variables: Declaration

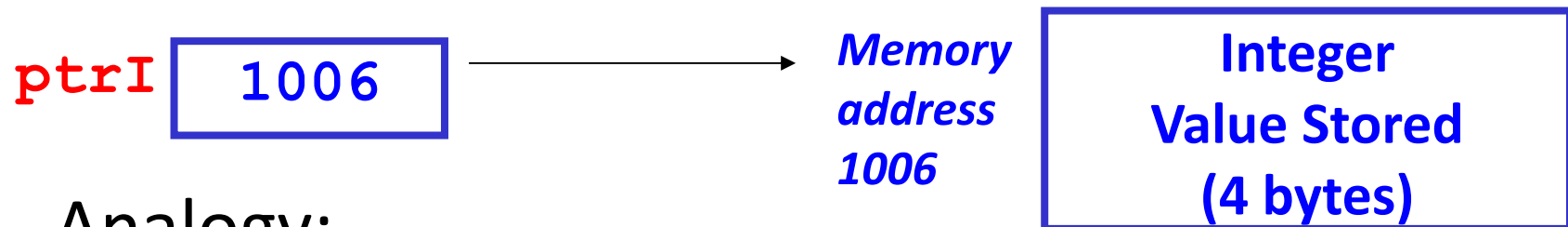
- **Pointer variable** – different from the primitive variable **num** (variable of primitive data type such as int, float, char) declared earlier, it stores the address of memory location of a data object.
- A **pointer variable** is declared by, for example:

```
int *ptr1;  
or  int * ptr1;  
or  int* ptr1;
```



- **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.

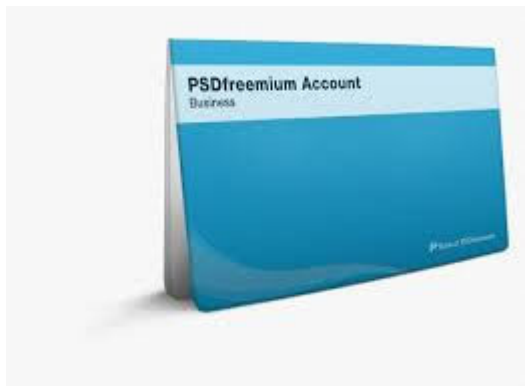
Pointer Variables: Analogy



- Analogy:
(1) Address on envelope \rightarrow your home



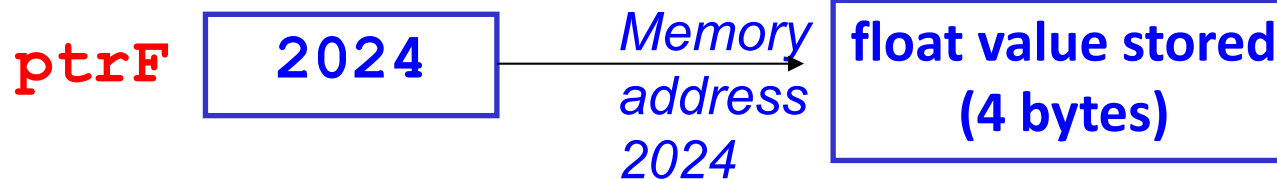
- (2) Bank account \rightarrow your saving/money in the bank



Pointer Variables: Declaration Examples

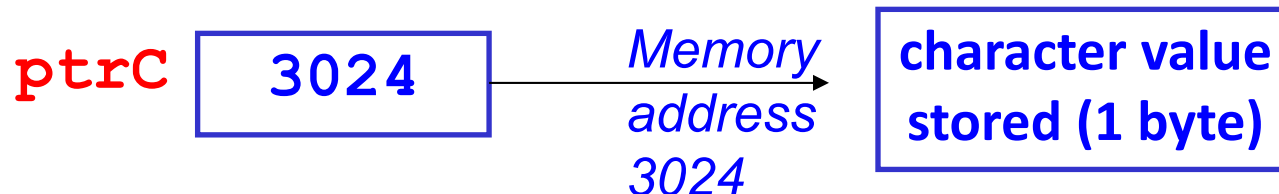
```
float *ptrF;
```

- **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.



Pointer Variables: Key Ideas

```
int * ptr1;
```

(1) ptr1

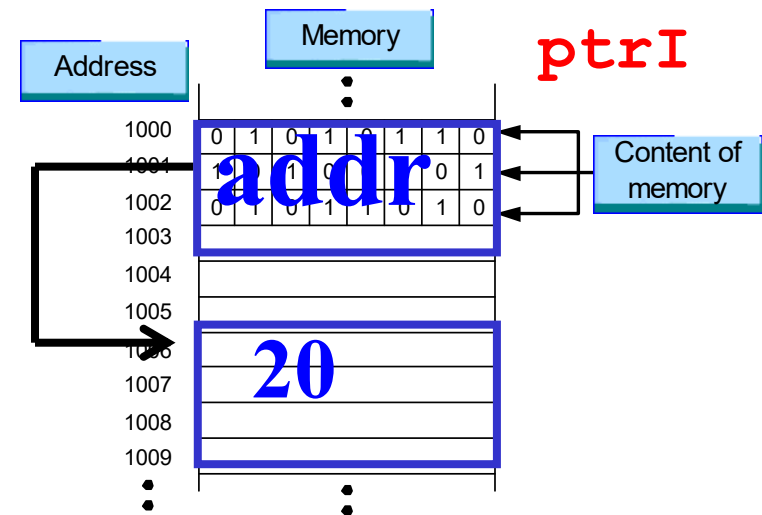
- Pointer variable (4 bytes of memory).
- The value of the variable (i.e. stored in the variable) is an address.

(2) *ptr1

- Contains the content (or value) of the memory location pointed to by the pointer variable ptr1.
- The value is referred to by using the indirection operator (*), i.e. *ptr1.
- For example: we can assign

***ptr1 = 20;**

=> the value 20 is stored at the address pointed to by ptr1.

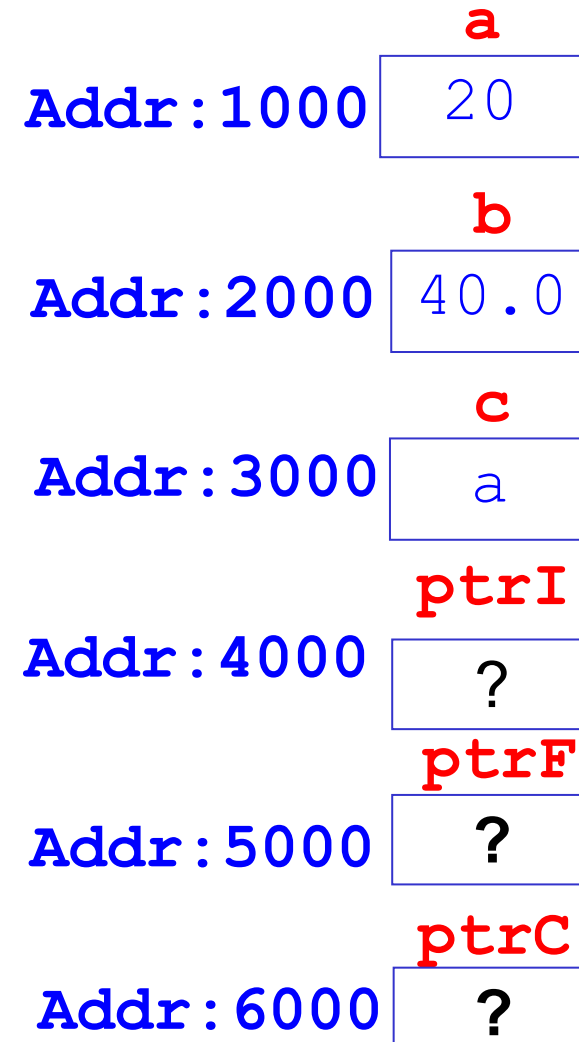


How to use Pointer Variables?

- **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.



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]

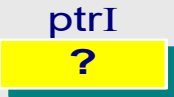



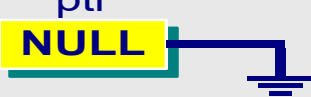
ptrC = &c; => *ptrC == 'a' [same as c]

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

Similarly,

*ptrF and b==40.0

*ptrC and c =='a'

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

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 = #	
*ptr = 10;	

Output



num = 3, &num = 1024

ptr = 1024, *ptr = 3

15 } **Note: num and *ptr have the same value**

Pointer Variables – Example 1 (Cont'd.)

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

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

```
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
```

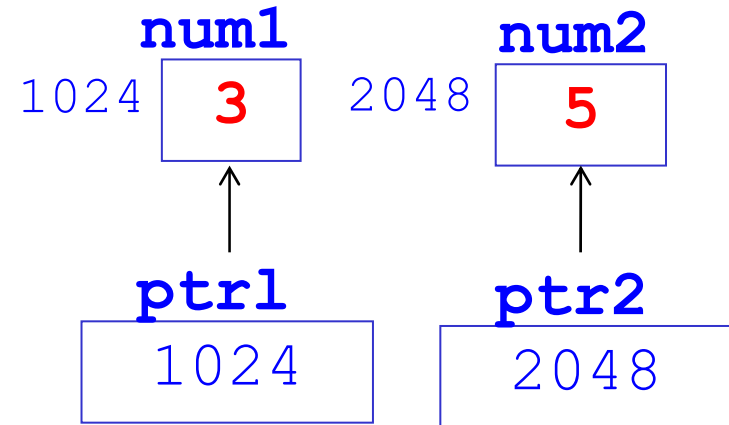
```
ptr = 1024, *ptr = 3
```

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

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


Pointer Variables – Example 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
```

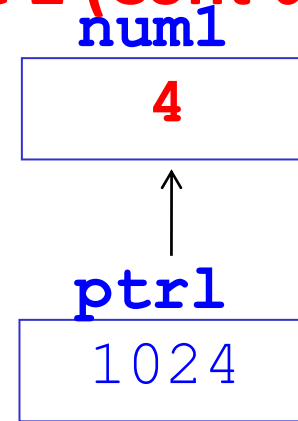
Pointer Variables – Example 2 (Cont'd.)

```
/* 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

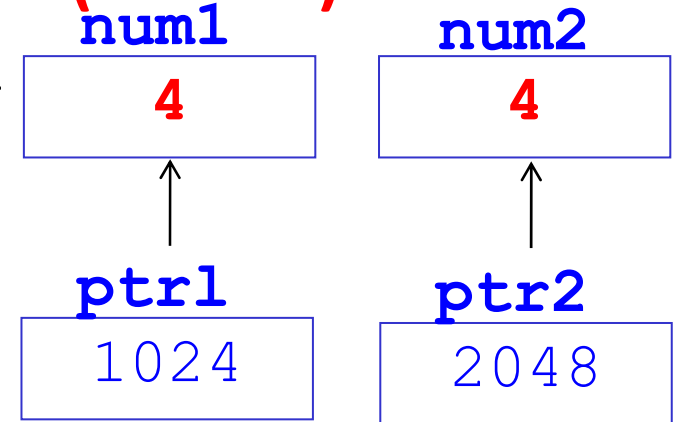
Pointer Variables – Example 2 (Cont'd.)

`/* 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

Pointer Variables – Example 2 (Cont'd.)

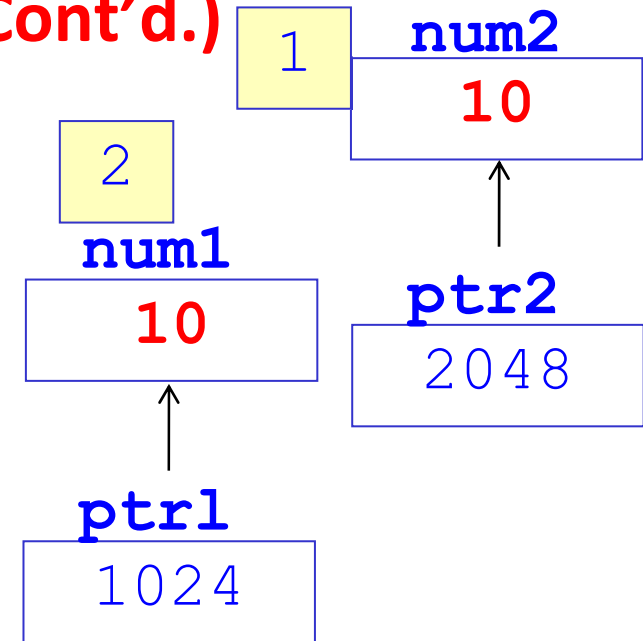
***ptr2 = 10;**

1 /* 10 copied into the location pointed by ptr2 */

num1 = *ptr2;

2 /* copy the content of the memory location pointed by ptr2 into num1 */

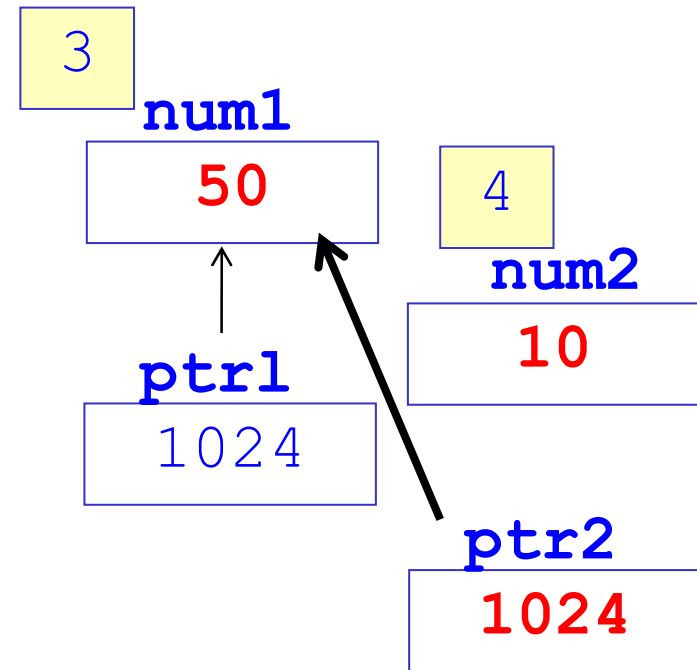
printf("num1 = %d,*ptr1 = %d\n",num1, ***ptr1**);



Output

num1 = 10, *ptr1 = 10

Pointer Variables – Example 2 (Cont'd.)



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

3

```
printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);
```

```
ptr2 = ptr1;
```

4

```
/* address in ptr1 copied into ptr2 */
```

Output

```
num1 = 50, *ptr1 = 50
```

```
num2 = 10, *ptr2 = 50
```

```
printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);
```

```
return 0;
```

```
}
```

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

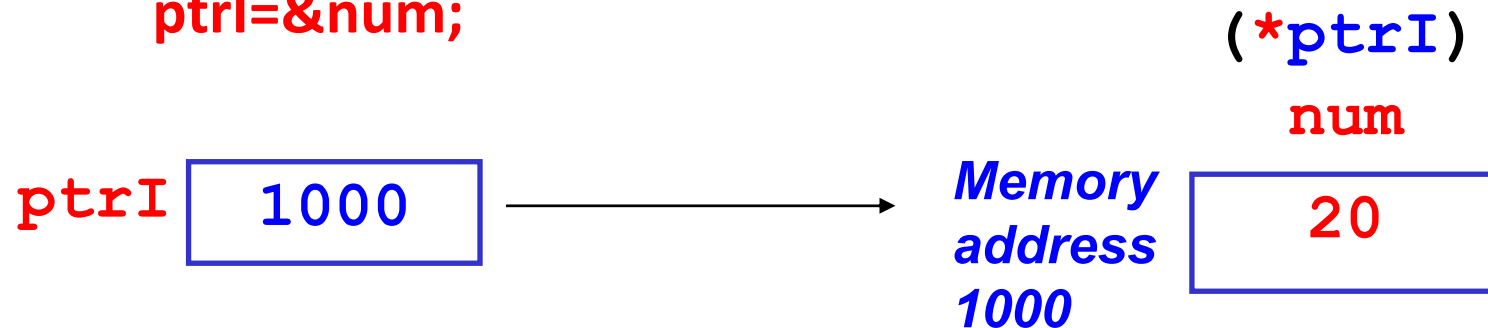
1. Declare variables and pointer variables:

```
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.

Pointers

- Primitive Data Types, Variables and Address Operator
- Pointer Variables
- **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);**

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);
```

Output
The value of num is: 6

```
int main( )
```

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

num 5 -> 6

```
int add1(int value)
```

// value – called parameter

```
{
    value++;
    return value;
}
```

value 5 -> 6

Call by Reference: Example 1

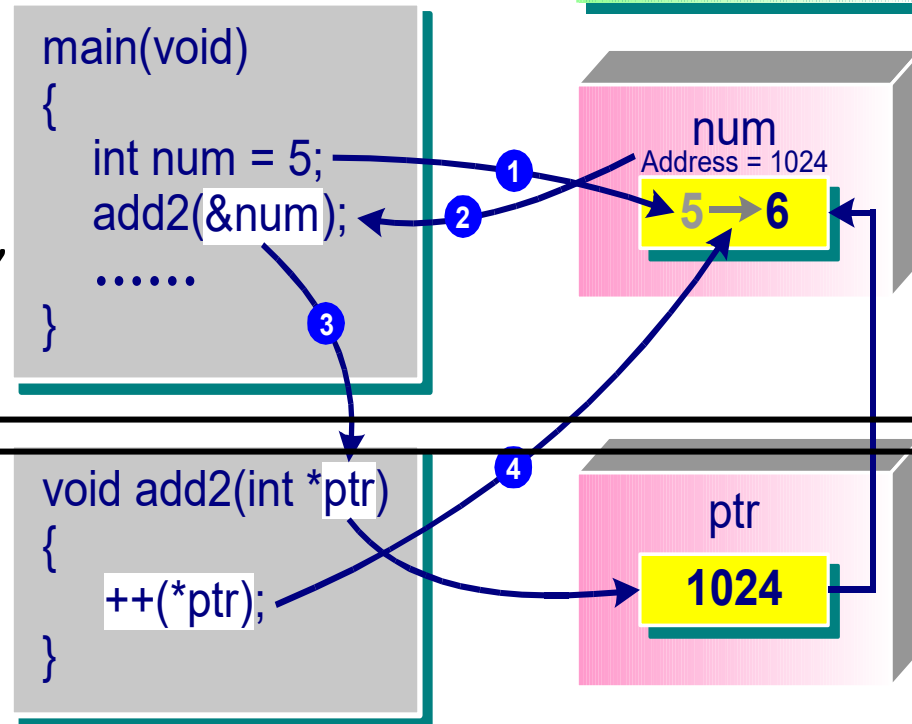
```
#include <stdio.h>
void add2(int *ptr);
int main()
```

Output
Value of num is 6

```
{
    int num = 5;
    /*passing the address of num*/
    add2(&num);
    printf("Value of num is: %d",
           num);
    return 0;
}
```

```
void add2(int *ptr)
```

```
{
    ++(*ptr);
}
```



- Any change to the value pointed to by the parameter **ptr** will **change** the argument value **num** (instantly).

Call by Reference: Key Steps

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

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

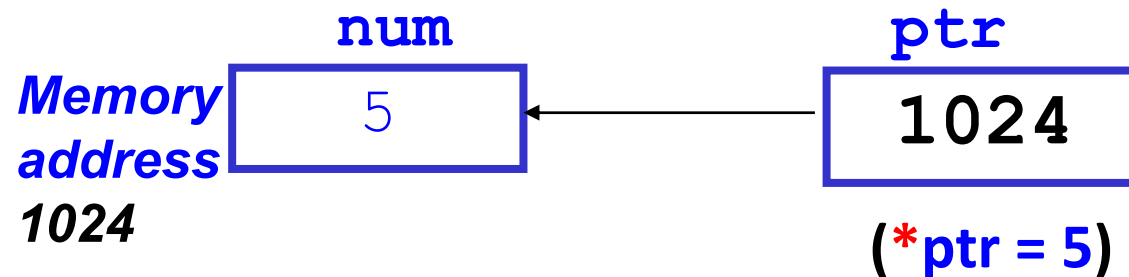
2. In the calling function, the arguments must be pointers (or using **address** operator as the prefix):

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

Call by Reference: Analogy

Communications between 2 functions: Call by Reference

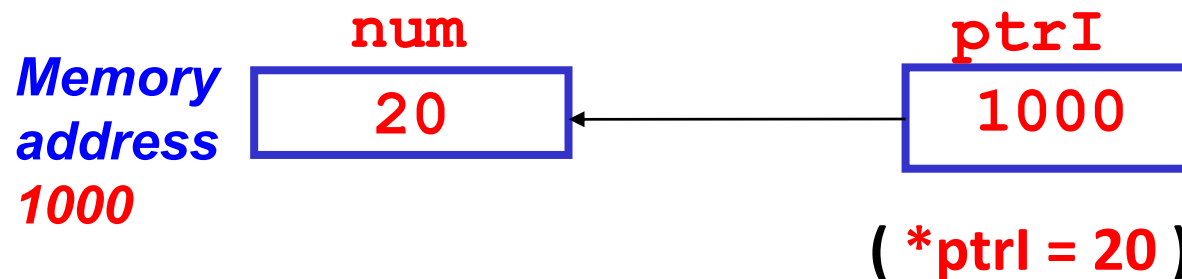
(1) **main()**: `int num; add2(&num);` (2) `void add2(int *ptr){...}`



Analogy: using pointer within a function:

(1) `int num; int *ptr1;`

(2) `ptr1 = #`



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;
}
```

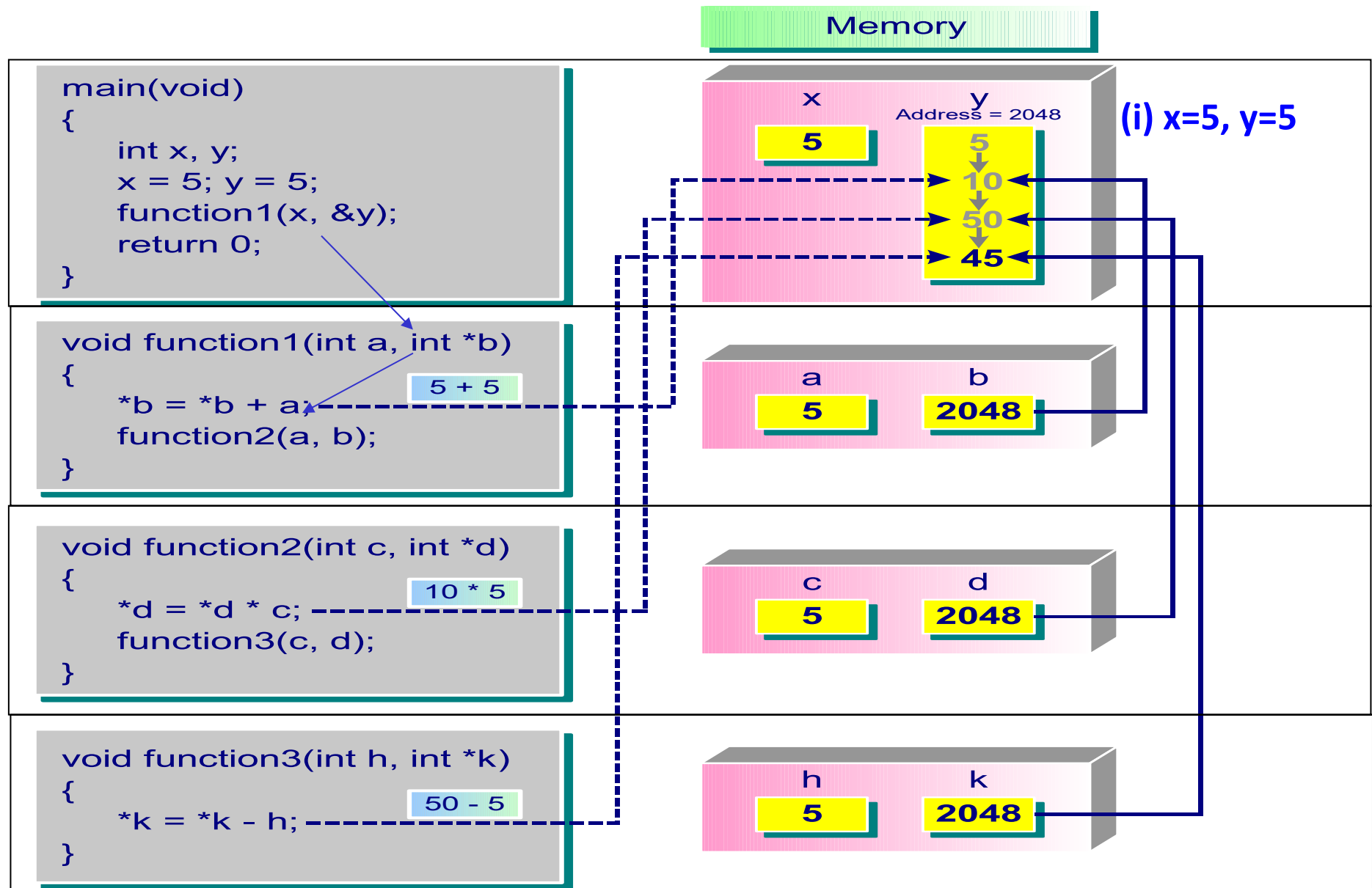
Diagram annotations:

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

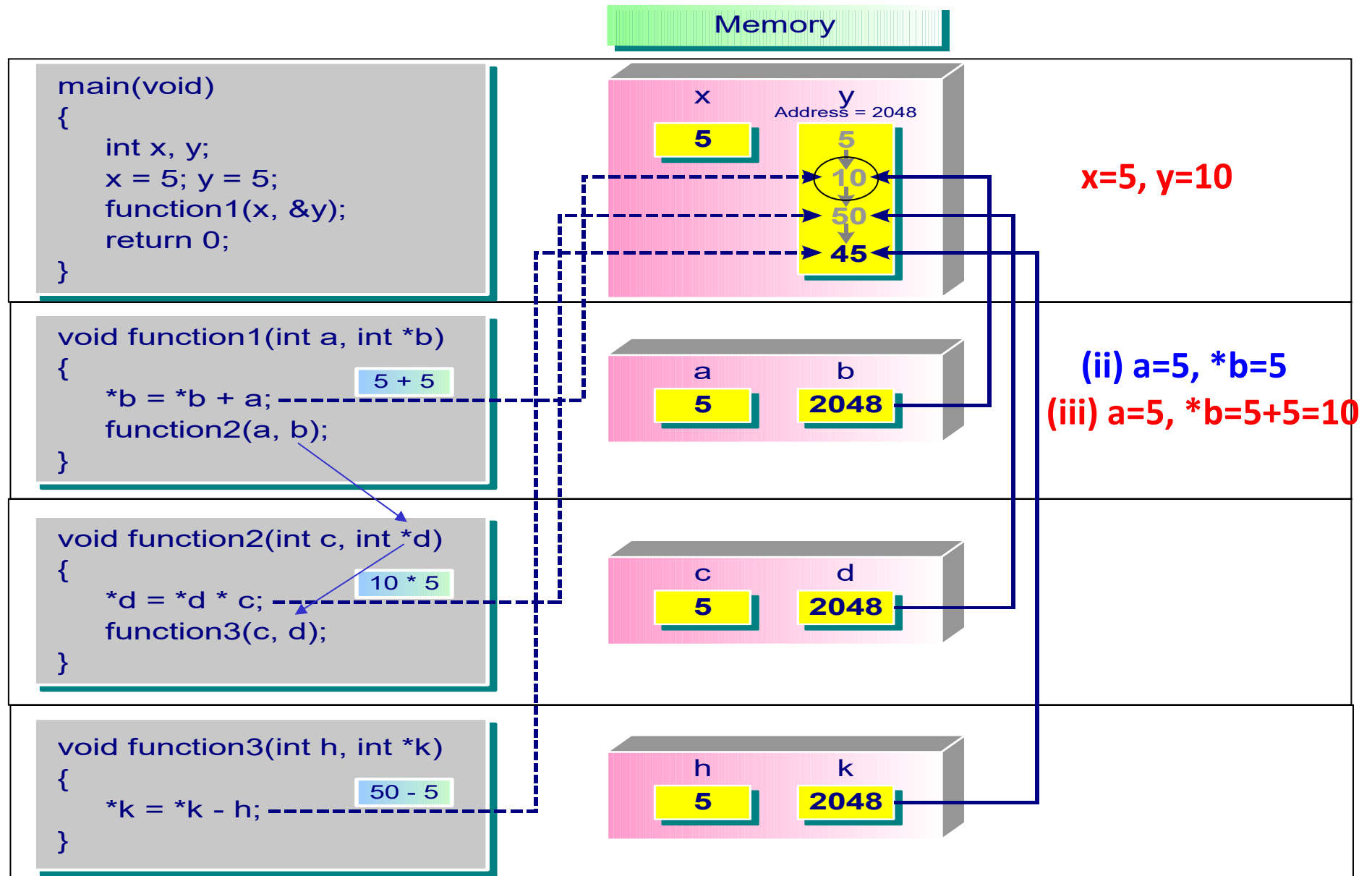
Comments on the right side of the code:

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

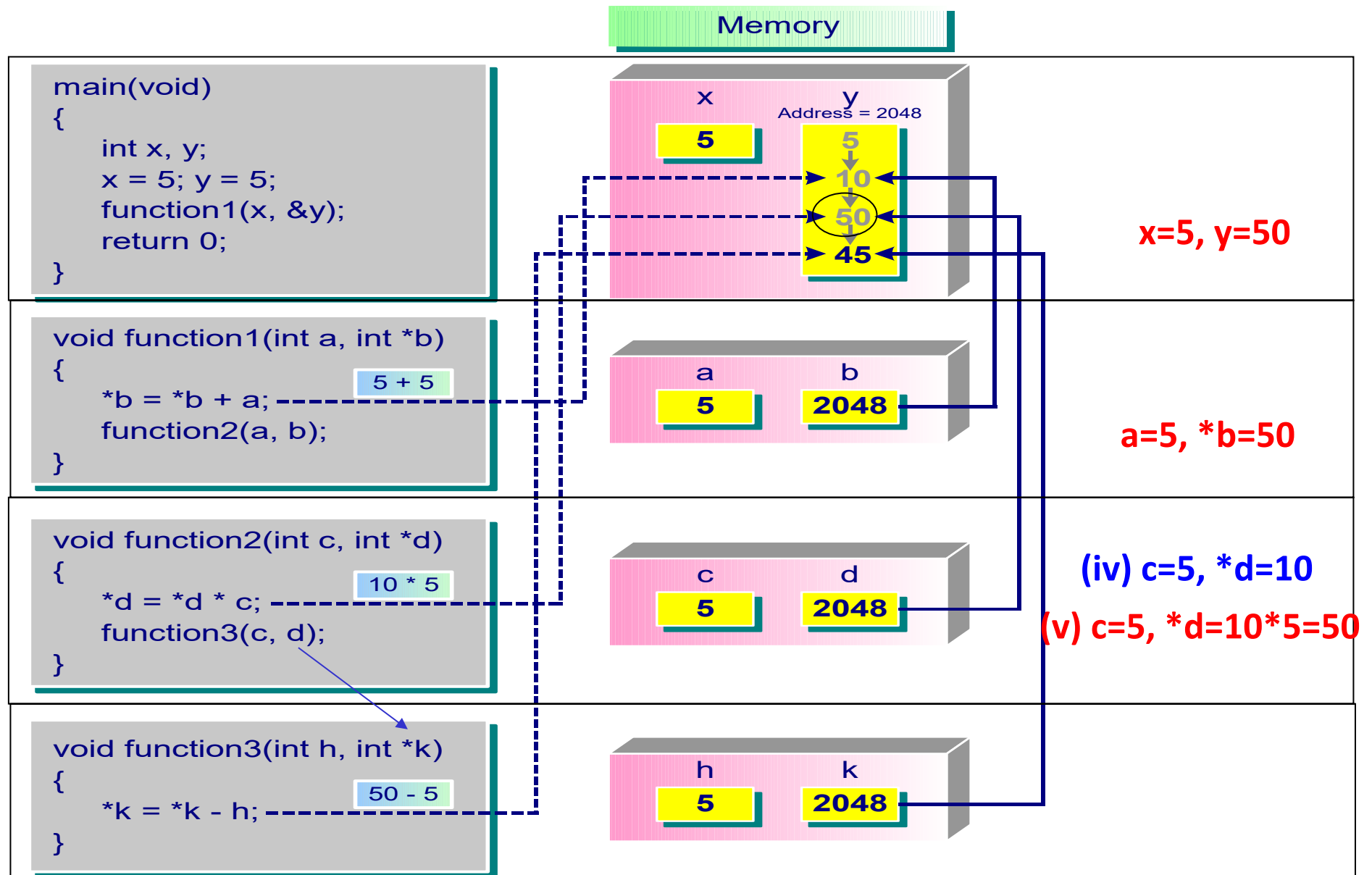
Call by Reference – main()



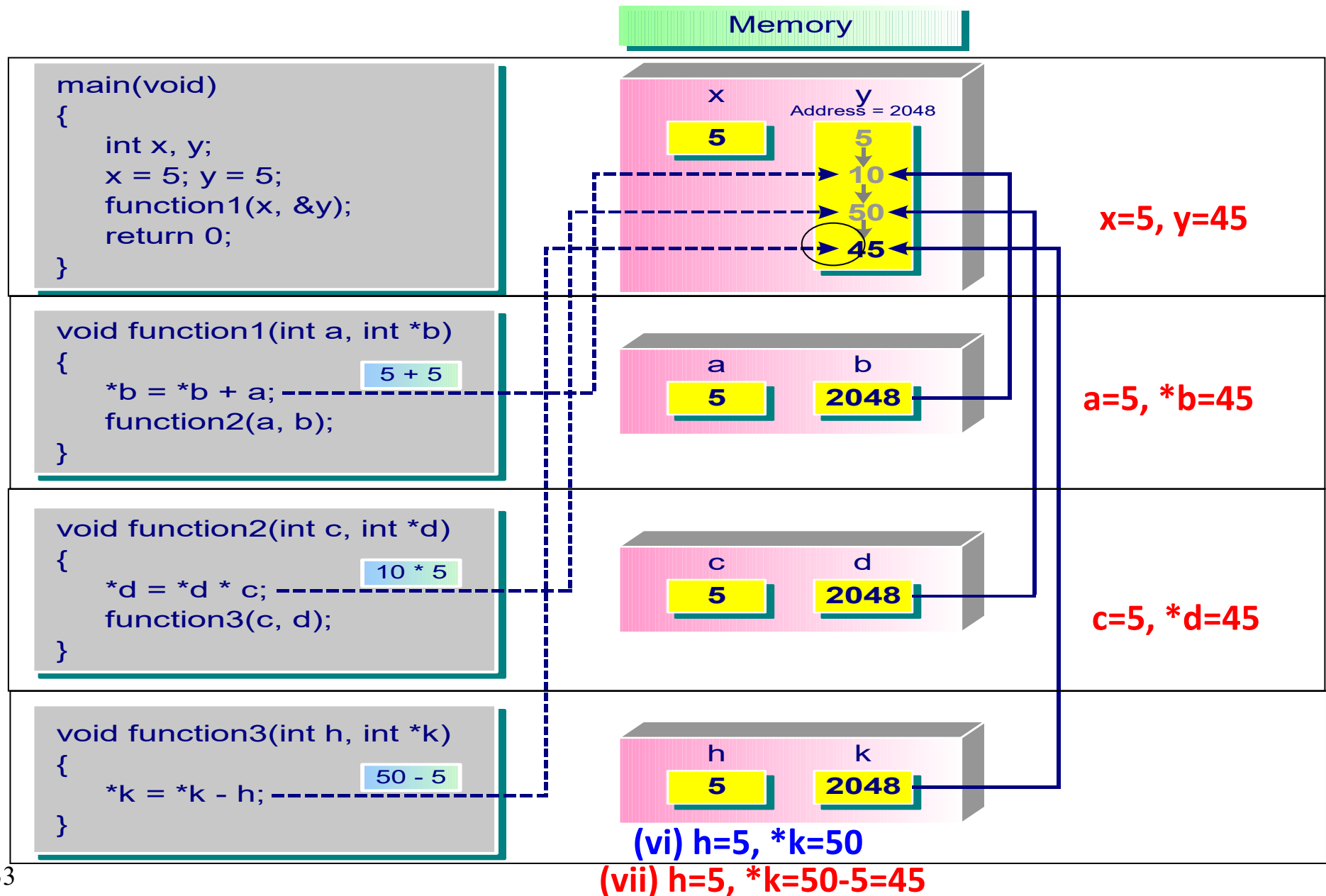
Call by Reference – function1()



Call by Reference – function2()



Call by Reference – function3()



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

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.

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;
```

```
}
```

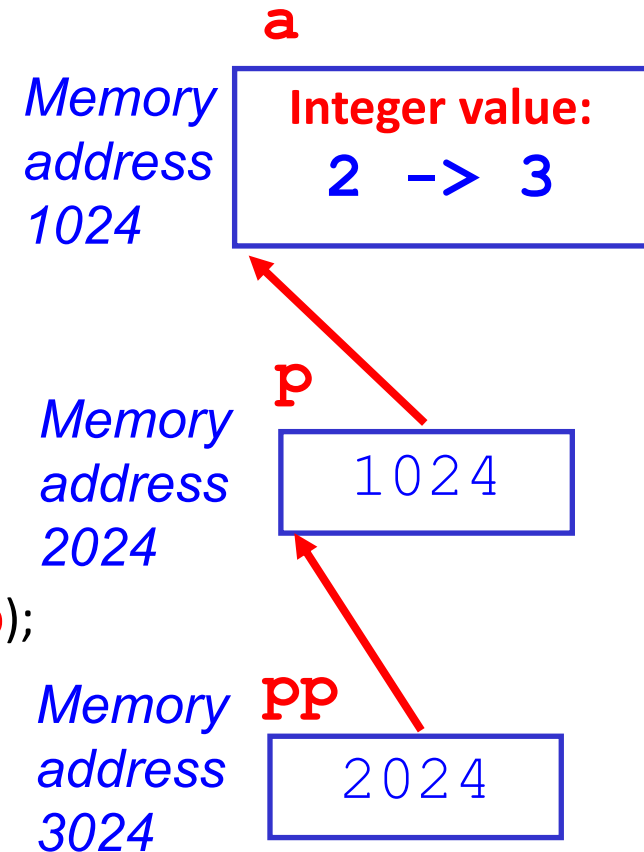
Output

```
a = 3
```

```
*p = 3
```

```
**pp = 3
```

Note: it could also be `int ***ppp;`
etc. The idea remains the same.



Thank You!