Developing Technical Software

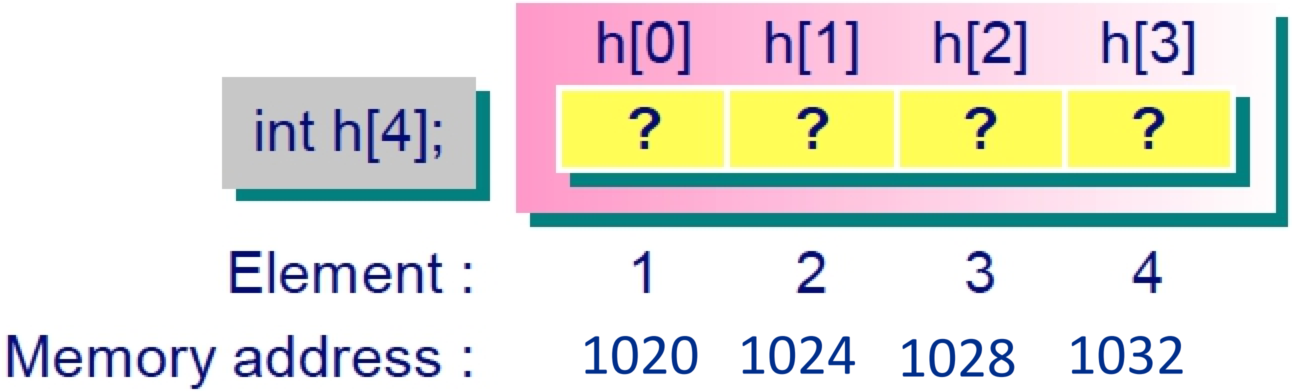
Week 2:
Revisit Technical Programming –
Structured programming with functions: passing parameters by value and by reference

SWINBURNE UNIVERSITY OF TECHNOLOGY

Recall: Arrays & Pointers

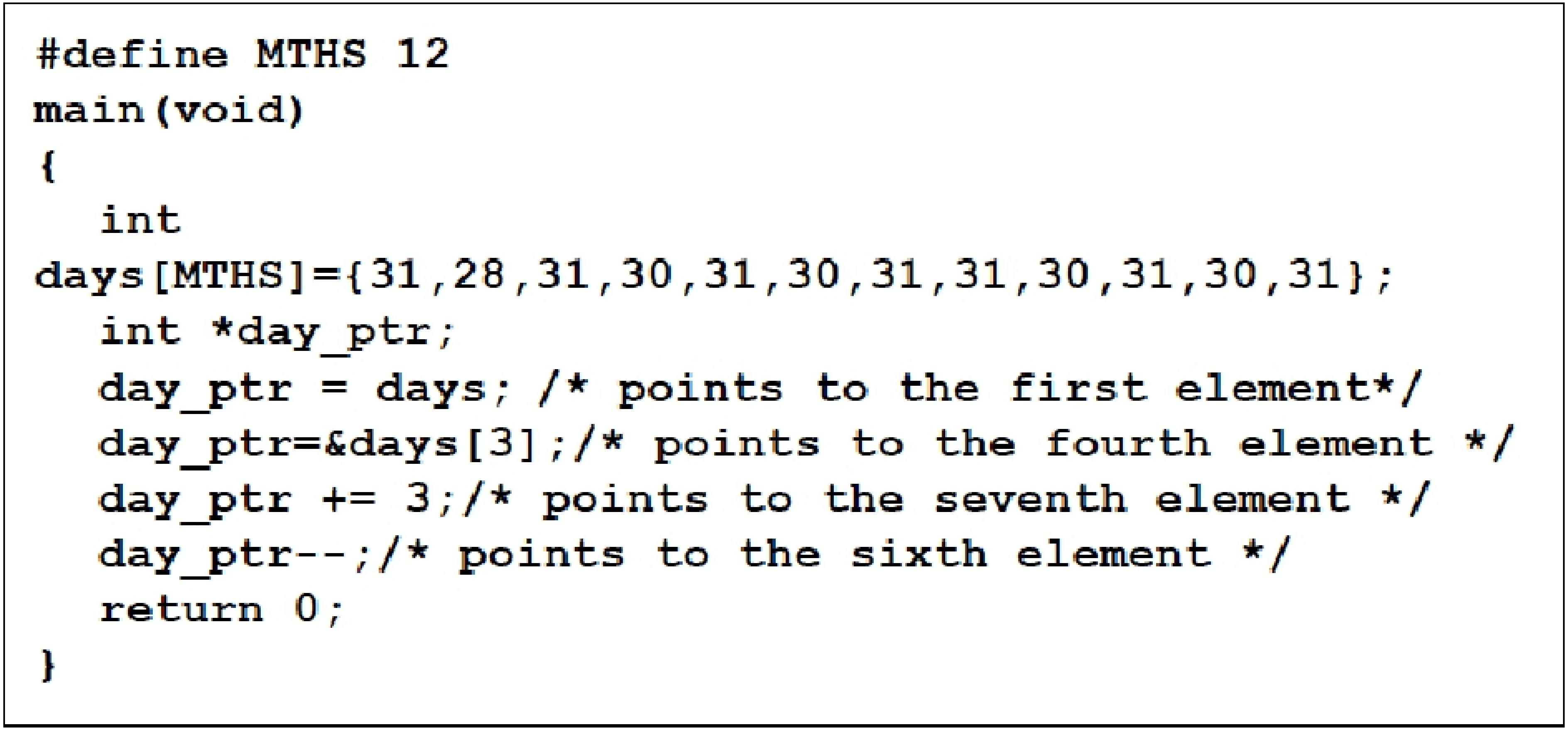
- An array is a collection of variables of the **same** type.
- When an array is declared, n consecutive memory locations are allocated by the compiler (index range from **0** to **n-1**).

```
int h[4];
```

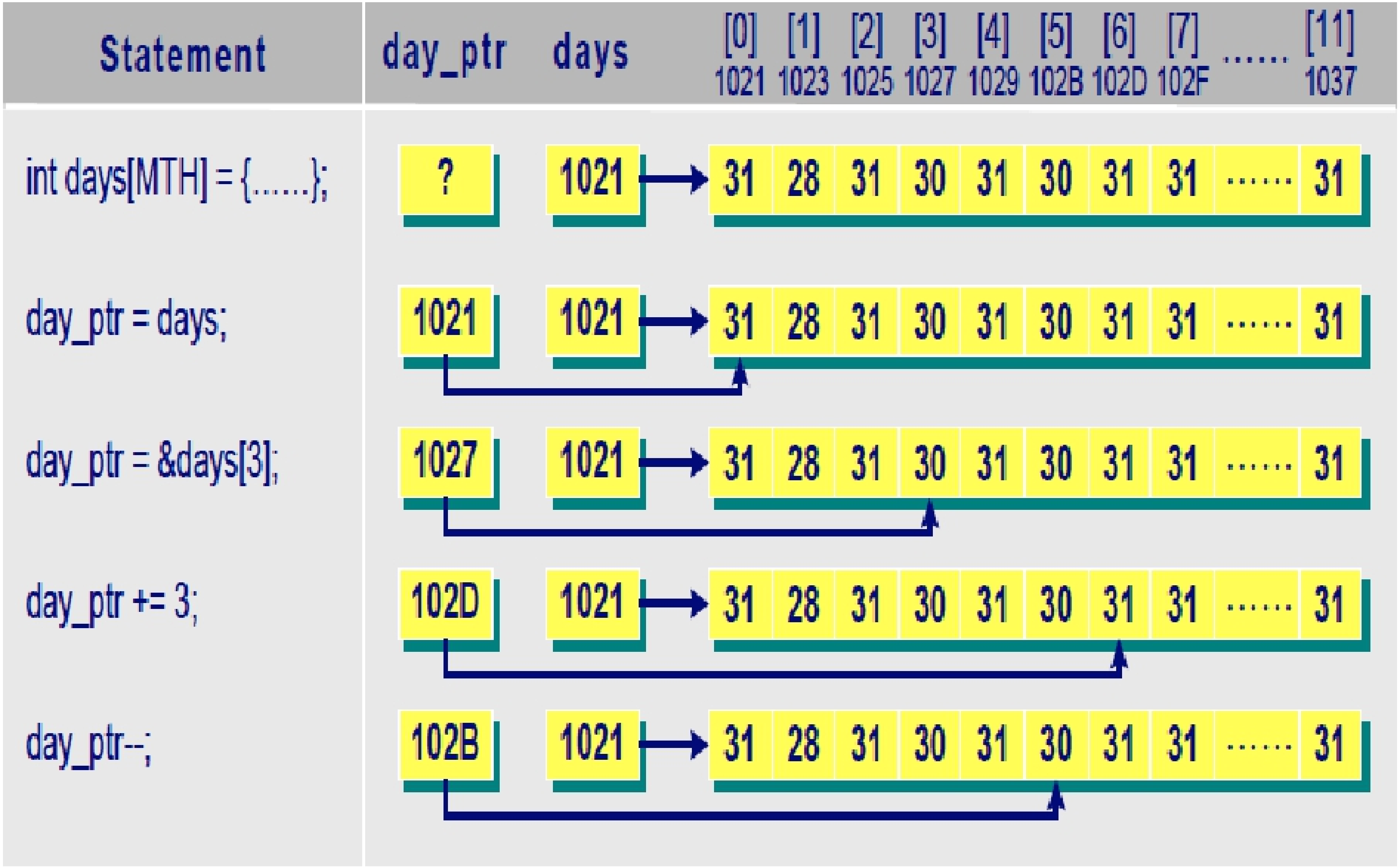


- The array name is a **constant** pointer to the first element of the array.
- Pointers can be subscripted like array. For the above example, if we have `hPtr=h`, then `hPtr[1]` refers to the array element `h[1]`.

Pointers and arrays

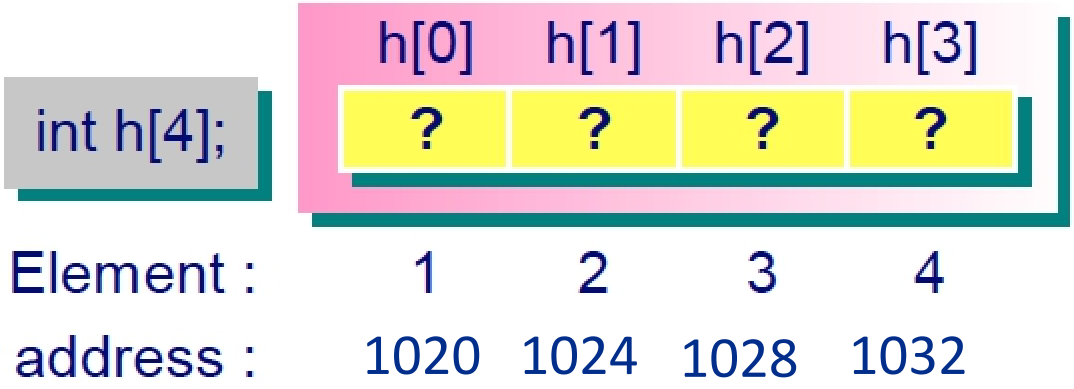


```
#define MTHS 12
main(void)
{
   int
days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
   int *day_ptr;
   day_ptr = days; /* points to the first element*/
   day_ptr=&days[3];/* points to the fourth element */
   day_ptr += 3;/* points to the seventh element */
   day_ptr--;/* points to the sixth element */
   return 0;
}
```



Statement	day_ptr	days	[0] 1021	[1] 1023	[2] 1025	[3] 1027	[4] 1029	[5] 102B	[6] 102D	[7] 102F	......	[11] 1037
int days[MTH] = {......};	?	1021 →	31	28	31	30	31	30	31	31	......	31
day_ptr = days;	1021	1021 →	31	28	31	30	31	30	31	31	......	31
day_ptr = &days[3];	1027	1021 →	31	28	31	30	31	30	31	31	......	31
day_ptr += 3;	102D	1021 →	31	28	31	30	31	30	31	31	......	31
day_ptr--;	102B	1021 →	31	28	31	30	31	30	31	31	......	31

Recall: Arrays & Pointers



int h[4];

	h[0]	h[1]	h[2]	h[3]
	?	?	?	?
Element :	1	2	3	4
Memory address :	1020	1024	1028	1032

- For an element in an array ***h[i]*** at index ***i***, we can retrieve the address of a particular element in the memory using either

  &h[i] or simply ***(h+i)***

- The value of the array ***h[i]*** at index ***i*** can be retrieved using

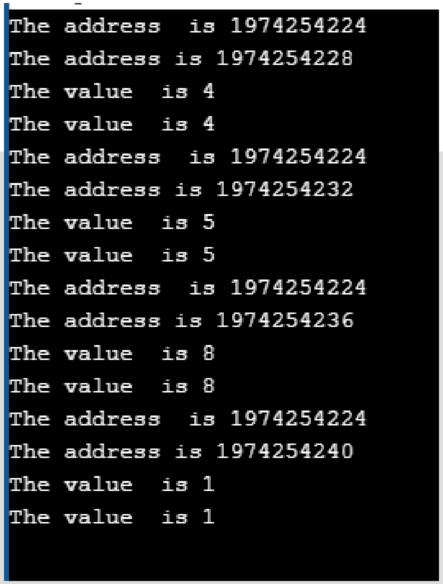
  h[i] or ****(h+i)***

- Never use * in front of array to dereference - ****h[i]***

Expected Output?

```c
#include <stdio.h>
int main()
{
int A[5] = {10, 4, 5, 8, 1};
int i;
for(i=0; i<5; i++)
 {
 printf("The address  is %d \n", &A[0]);//Prints the address of respective element of the array
 printf("The address is %d \n", A+i);//Prints the address of respective element of the array
 printf("The value  is %d\n", A[i]);//Prints the value of the respective address of the element
 printf("The value  is %d \n", *(A+i));//Prints the value of the respective address of the element
 }
return 0;
}
```

Expected Output?

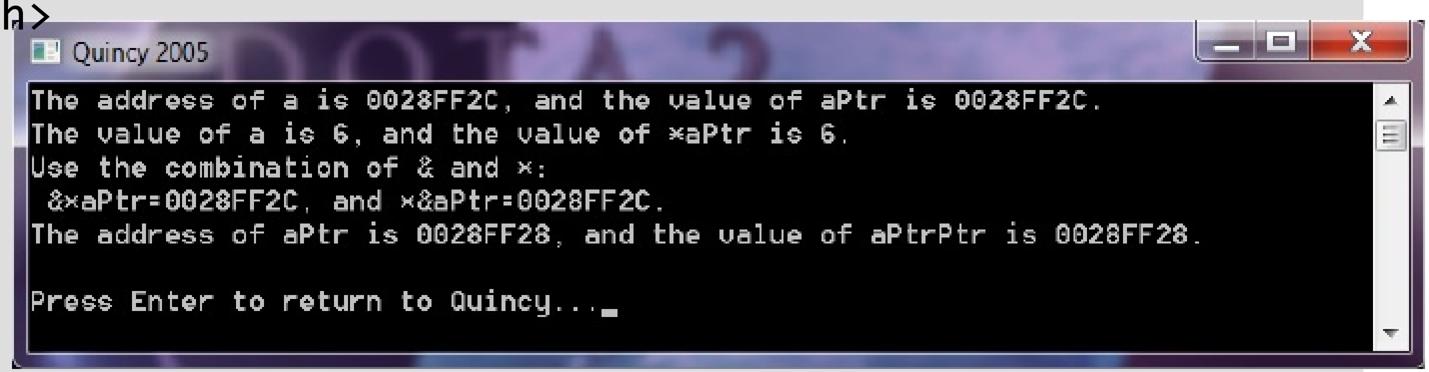


```
The address  is 1974254224
The address is 1974254228
The value  is 4
The value  is 4
The address  is 1974254224
The address is 1974254232
The value  is 5
The value  is 5
The address  is 1974254224
The address is 1974254236
The value  is 8
The value  is 8
The address  is 1974254224
The address is 1974254240
The value  is 1
The value  is 1
```

```
#include <stdio.h>
int main()
{
int A[5] = {10, 4, 5, 8, 1};
int i;
for(i=0; i<5; i++)
 {
 printf("The address  is %d \n", &A[0]);//Prints the address of the respective element of the array
 printf("The address is %d \n", A+i);//Prints the address of the respective element of the array
 printf("The value  is %d\n", A[i]);//Prints the value of the respective address of the element
 printf("The value  is %d \n", *(A+i));//Prints the value of the respective address of the element
 }
return 0;
}
```

Expected Output?

```c
#include <stdio.h>
int main()
{
   int a, *aPtr;
   int **aPtrPtr;
   a=6;
   aPtr=&a;
   aPtrPtr=&aPtr;
   printf("The address of a is %p, and the value of aPtr is %p.\n",
          &a, aPtr);
   printf("The value of a is %d, and the value of *aPtr is %d.\n",
          a, *aPtr);
   printf("Use the combination of & and *:\n &*aPtr=%p, and
          *&aPtr=%p.\n", &*aPtr, *&aPtr);
   printf("The address of aPtr is %p, and the value of aPtrPtr is
          %p.\n", &aPtr, aPtrPtr);
   return 0;
}
```



```
The address of a is 0028FF2C, and the value of aPtr is 0028FF2C.
The value of a is 6, and the value of *aPtr is 6.
Use the combination of & and *:
 &*aPtr=0028FF2C, and *&aPtr=0028FF2C.
The address of aPtr is 0028FF28, and the value of aPtrPtr is 0028FF28.

Press Enter to return to Quincy..._
```

Programmer-Defined Functions

Example

```
#include <stdio.h>
float pi (void); /* function prototype */

int main ( )
{
    float p;
    p = pi( );
    printf("The value of pi is %f \n", p);
    return 0;
}

float pi (void)
{
   return 3.141593;
}
```

```
The value of pi is
3.141593
```

- **A function returns a value except a void function.**

Programmer-Defined Functions

- **Function Prototype**

```
float pi (void);
```

It informs the compiler that the main function will reference a function named pi

- **Function Definition**

```
return_type  func_name(para_declarations)
{
  statements;
  return 3.141593;
}
```

```
float pi (void)
{
   return 3.141593;
}
```

- **Function Call** `p = pi( );`

The execution of a program always begins with the main function. The programmer-defined function is called when the program encounters the function name pi

Recursion

- A **recursive function** is a function that *calls itself* either directly or indirectly through another function.
- We may consider solving a problem by recursive function call if
  - The *simplest case* of the problem could be easily solved (**base case**)
  - The problem could be represented by a *similar but simpler* version of the original problem (**recursive call**)
- **Example**: Recursively calculating factorials

```
long fact(int n)
{
     if  (n<=1) return 1;
     else return (n*fact(n-1));
}
```

With vs. Without programmer-defined function

Example: Find the maximum number of three integers

```c
#include <stdio.h>
int max(int, int);

int main( )
{ int a, b, c;
  printf(”Input three integers \n”);
  scanf(“%d%d%d”,&a,&b,&c);
  printf(“The maximum number is:”);
  printf(“%d”,max(max(a,b),max(a,c)))
  return 0;
}

int max(int x, int y)
{
  if (x >= y) return x;
  else return y;
}
```

With programmer-defined function

```c
#include <stdio.h>
int main( )
{int a, b, c, max;
  printf(”Input three integers \n”);
  scanf(“%d%d%d”,&a,&b,&c);

  if (a > b && a > c)
    max = a;
  else if (b > c && b > a)
    max = b;
  else
    max = c;

  printf(“The maximum is: %d”,max);
  return 0;
}
```

Without programmer-defined function

Pass parameter(s) to the function

Suppose we hope the function `pi` returns an integer multiple of the value π (e.g., 2π), the multiplier could be specified by a parameter as follows:

Example

```
#include <stdio.h>
float pi(int);

int main( )
{
    float p;
    p = pi(2);
    printf(“The value of two pi is %f \n”, p);
    return 0;
}
float pi(int n)
{
    return 3.141593*n;
}
```

```
The value of  two pi is
6.283186
```

Local Variables

- **Local variables** are defined within a function, thus include the formal parameters and any other variables declared in the function. In the example result, a, b inside the main function are local variables
- **Local variables** can be accessed only in the function that defines them.
- **A local variable** has a value when its function is being executed, but its value is not retained when the function is completed.

Global Variables

- **Global variables** are defined **outside** the `main` function or other programmer-defined functions. In example `a, b` are global variables.
- **Global variables** can be accessed by any function within the program.

Global variables and Local variables

```
#include <stdio.h>
//Global variables
int a;
int b;

int add();
{
    return a+b;
}
int main( )
{
    int result; //Local variables
    a = 5, b = 7;
    result = add();
    printf(“%d\n”, result);
    return 0;
}
```

Calling functions by value and by reference

- **Call by value**: the formal parameter holds a copy of the actual parameter. Therefore, changes to the formal parameter in a function is done on the copy, not the actual parameter.
- **Call by reference**: the formal parameter holds the address of the actual parameter, i.e., the formal parameter is a pointer. Therefore, changes to the value pointed to by the formal parameter changes the actual parameter instantly.

Example: Parameter pass by value

2^8

```
int power(int x, int n)
  {
    int p;
    for (p=1; n>0; n--)
      p =p*x;
    return p;
   }
```

```
#include <stdio.h>
int power(int, int);
int main( )
{ int a, b, c;
  a=2;
  b=8;
  c=power(a,b);
  printf("The value of c is
              %3d", c);
  return 0;
}
int power(int x, int n)
{ int p;
  for (p=1; n>0; n--)
    p=p*x;
  return p;
}
```

- The value of n is altered within the function. but this does not change the value of b in the main body of the C program.
- Such a parameter passing method is called “**pass by value**”.

Parameter List

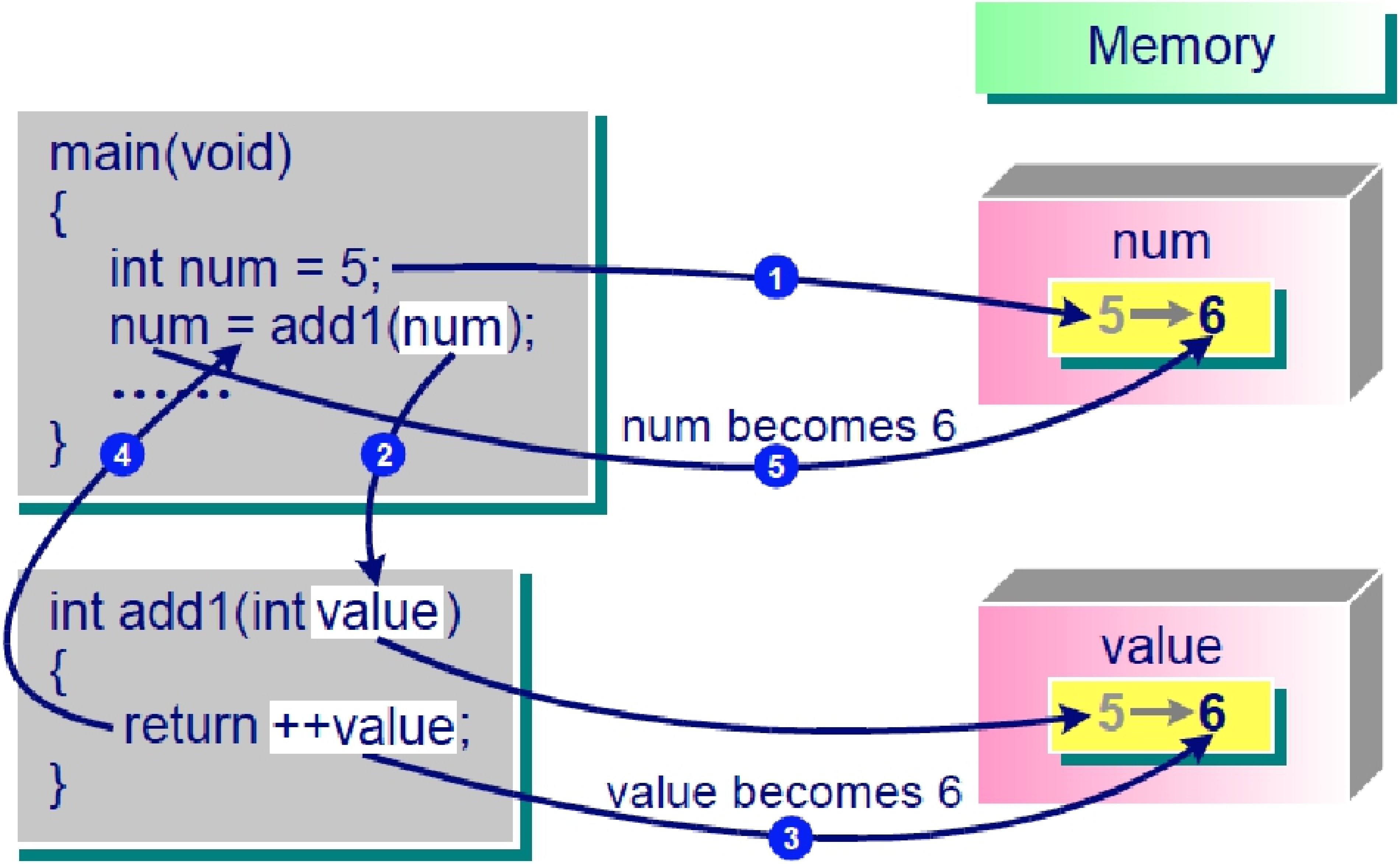
- The parameters `n, x` required by the function are called formal parameters, while `a, b` are actual parameters.
- The formal parameters and actual parameters **must match number, type, and order.**
- In the function call `power(a,b)`, the values in actual parameters are copied to the formal parameters, thus the values of actual parameters **are not altered by the function**.

```
#include <stdio.h>
int power(int, int);
int main( )
{ int a, b, c;
  a=2;
  b=8;
  c=power(a,b);
  printf("The value of c is
          %3d", c);
  return 0;
}
int power(int x, int n)
{ int p;
  for (p=1; n>0; n--)
    p=p*x;
  return p;
}
```

Another example for function call by value

Example

```
#include <stdio.h>
int add1(int);
int main( )
{
   int num = 5;
   num = add1(num);
   printf(“The value of num is: %d”, num);
   return 0;
}
int add1(int value)
{
   return ++value;
}
```



```
main(void)
{
   int num = 5;
   num = add1(num);
   .......
}
```

```
int add1(int value)
{
   return ++value;
}
```

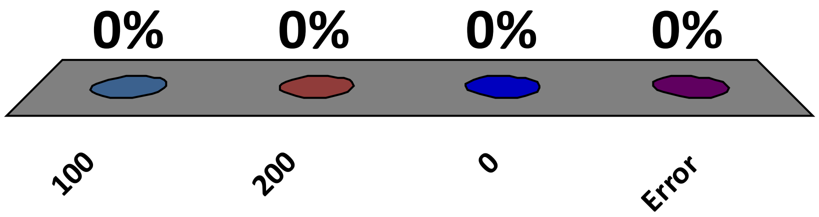
```
#include<stdio.h>
int sum(int a)
{
  a = a+100;
}
int  main()
{
int a=100;
sum(a);
printf("%d",a);
}
```

A. [✓ 100]

B. 200

C. 0

D. Error



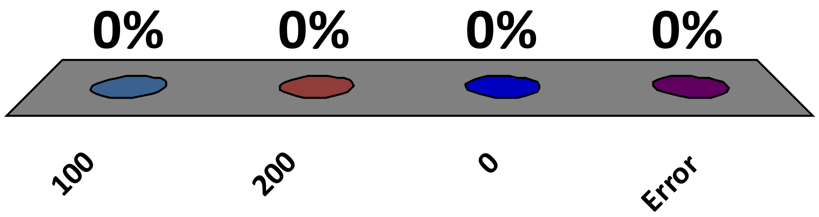
```
#include<stdio.h>
int sum(int a);
int  main()
{
int a=100;
sum(a);
printf("%d",a);
}

int sum(int a)
{
  return a = a+100;
}
```

A. [✓ 100]

B. 200

C. 0

D. Error



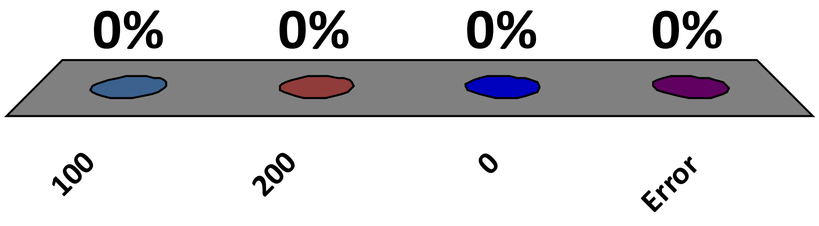
```
#include<stdio.h>
int sum(int a);
int  main()
{
int a=100;
a=sum(a);
printf("%d",a);
}


int sum(int a)
{
  a = a+100;
}
```

A. [✓ 100]

B. 200

C. 0

D. Error



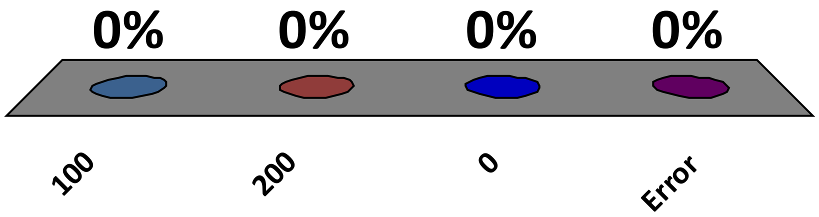
```
#include<stdio.h>
int sum(int a)
{
  a = a+100;
  return a;
}
int  main()
{
int a=100;
a=sum(a);
printf("%d",a);
}
```

A. 100

B. [✓ 200]

C. 0

D. Error



Evaluation order of function parameters

- It is never safe to depend on the order of evaluation of side effects, e.g. a function call like below may very well behave differently from one compiler to another:

```
void  func(int, int)

int i=2;
func (i++, i++);
```

- Either increment might happen first: func(2,2), or func(3, 2), or even func(3, 3)

Evaluation order of operands

- Consider the following piece of program, what would be the output of the program? ‘5’ or ‘10’ ?
- The output is undefined as the order of evaluation of `f1() + f2()` is not mandated by standard. The compiler is free to first call either `f1()` or `f2()`.

```
include<stdio.h>
int x = 0;

int f1()
{
  x = 5;
  return x;
}

int f2()
{
  x = 10;
  return x;
}

int main()
{
  int p = f1() + f2();
  printf("%d ", x);
  return 0;
}
```

Passing arrays to functions

- An array name is really the address of the first element of the array

```
#include <stdio.h>
int main()
{
 char array[5];
 printf(“  array=%p\n&array[0]=%p\n  &array=
               %p\n”, array, &array[0],&array);
}
```

```
   array =0012FF78
&array[0]=0012FF78
  &array=0012FF78
```

Passing arrays to functions (cont.)

- C automatically passes arrays to functions by reference

```
int a1[size];
modifyarray(int b[], int x);
modifyelement(int e);
```

- The function call `modifyarray(a1, size)` passes array `a1` and its size to function `modifyarray`

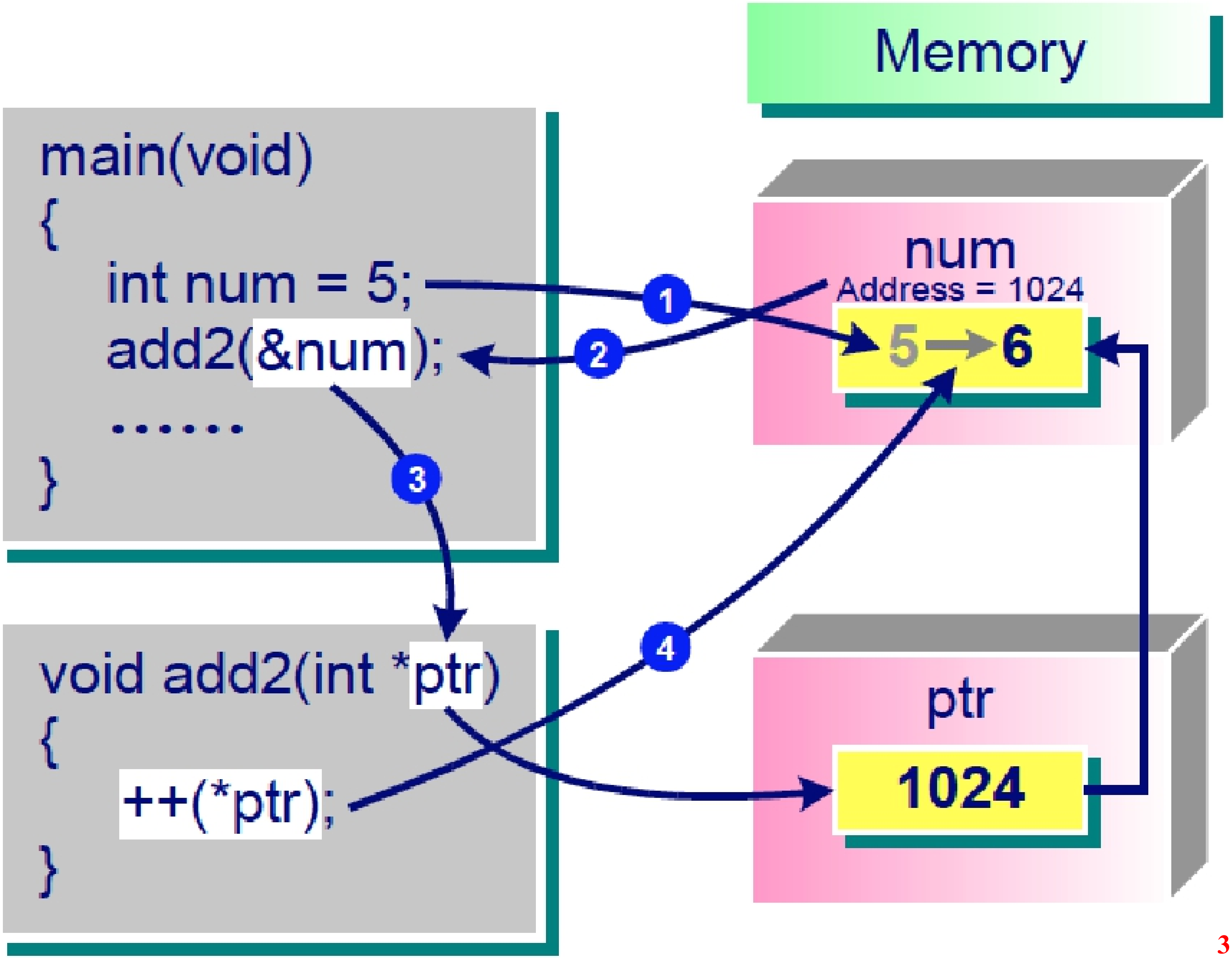
- There is a difference between passing an entire array and passing an array element. Compare the function calls below:

```
modifyarray(a1, size);
modifyelement(a1[0]);
```

Pointers - Function Call by References

Example

```
#include <stdio.h>
void add2(int *ptr);
int main( )
{
    int num = 5;
    add2(&num);
    printf(“The value of num is: %d”, num);
    return 0;
}
void add2(int *ptr)
{
    ++(*ptr);
}
```

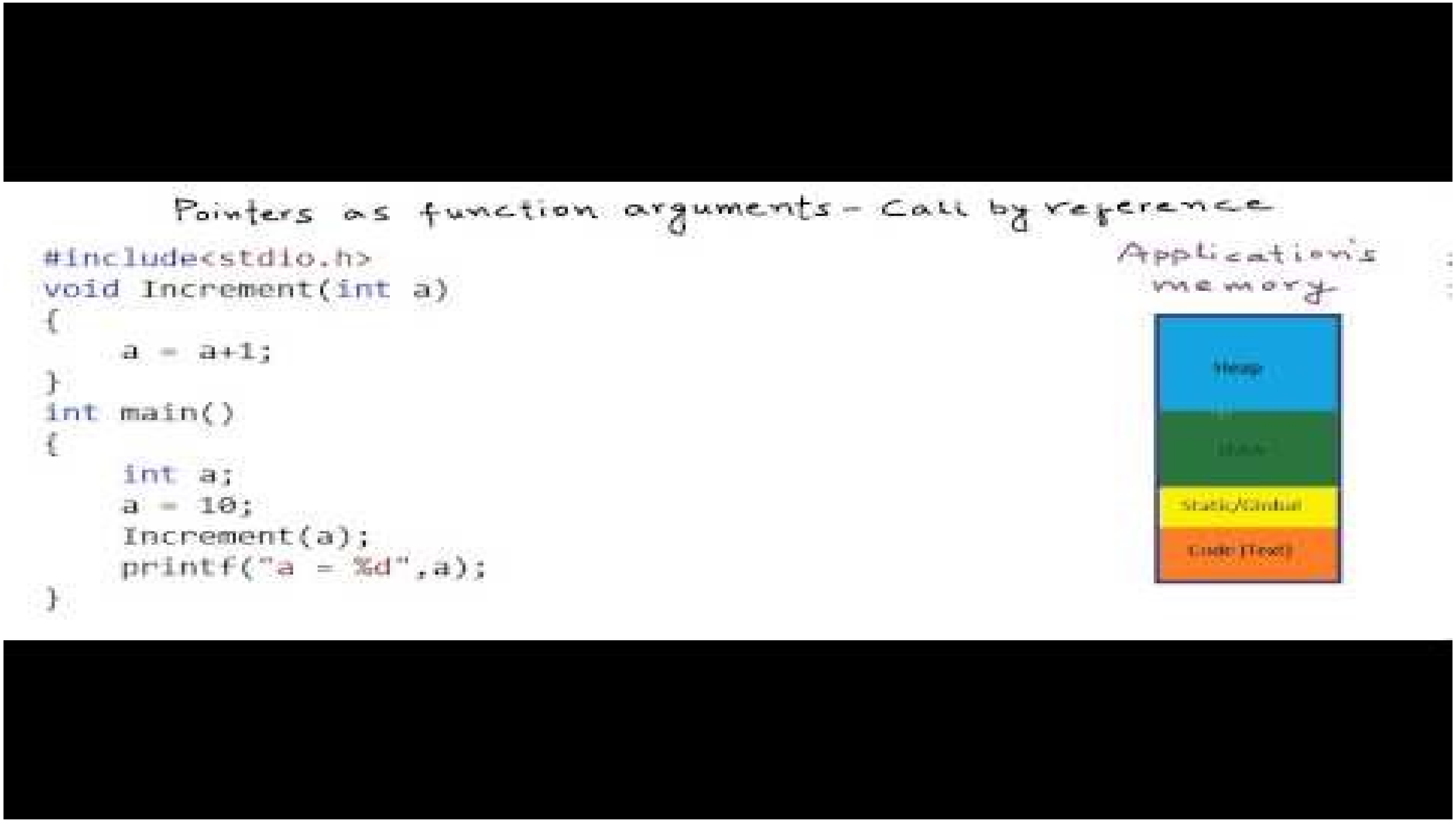


Memory

```
main(void)
{
    int num = 5;
    add2(&num);
    ......
}
```

```
void add2(int *ptr)
{
    ++(*ptr);
}
```

Pointers as function arguments



Pointers as function arguments - Call by reference

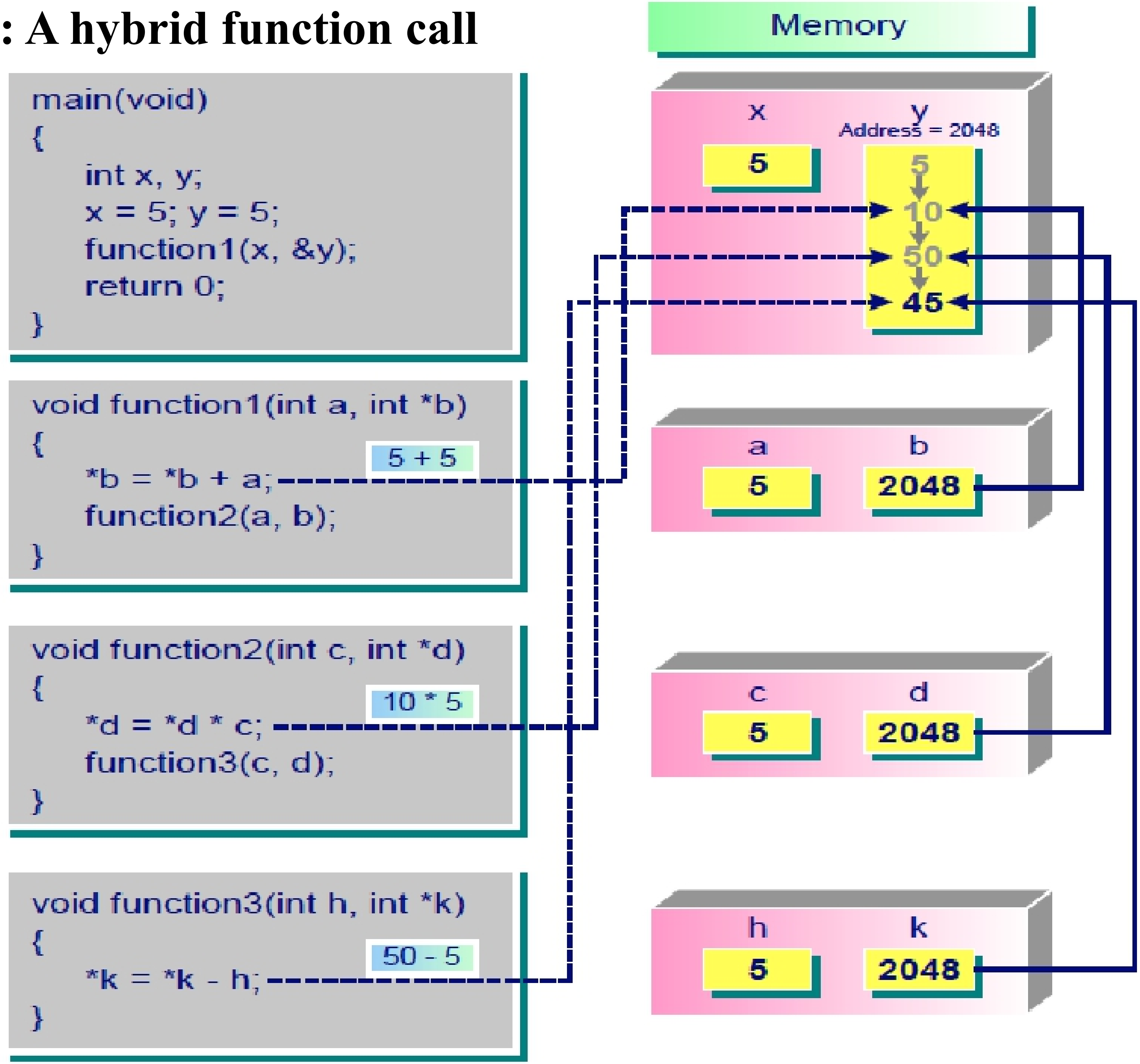
```
#include<stdio.h>
void Increment(int a)
{
     a = a+1;
}
int main()
{
     int a;
     a = 10;
     Increment(a);
     printf("a = %d",a);
}
```

https://www.youtube.com/watch?v=LW8Rfh6TzGg&index=5&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_

When to use call by reference?

- Need to pass more than one value back from a function
- Or if using call by value will result in a large piece of information being copied to the formal parameter (for efficiency)
- Call by reference saves a lot of memory – creating a copy of complex data type we can just use a reference to it

Example: A hybrid function call



```
main(void)
{
     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;
}
```

Thank you