Chapter 11

Pointer Applications

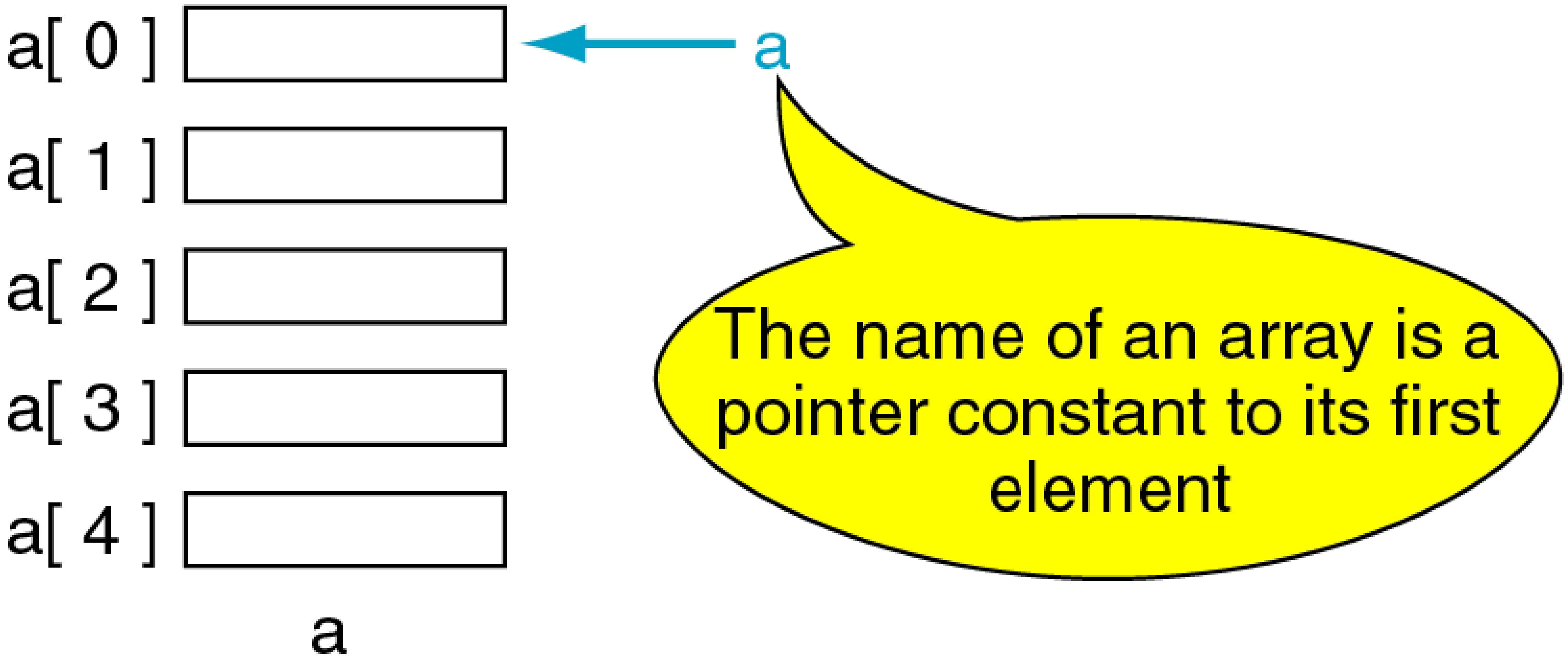
Pointer Applications

1. Arrays
2. Dynamic Memory

Pointer Applications

- Arrays and Pointers
- The name of an array is a pointer constant to the first element
- Since the array name is a pointer constant to the first element
  - its value cannot be changed
  - the address of the first element and the name of the array both represent the same location in memory
  - we can use the array name anywhere we can use a pointer, as long as it is used as an rvalue
  - we can use it with the indirection operator

Figure 11-1 Pointer to Arrays



The name of an array is a pointer constant to its first element

Arrays and Pointers

- When we dereference an array name . . .
- we are dereferencing the first element of the array & not the whole array
- a is same as &a[0]
- Program: demonstrate that array name is a pointer constant
- Print the address of the first element of the array and the array name

```
int a[5];
printf(“%p %p”, &a[0], a);
```

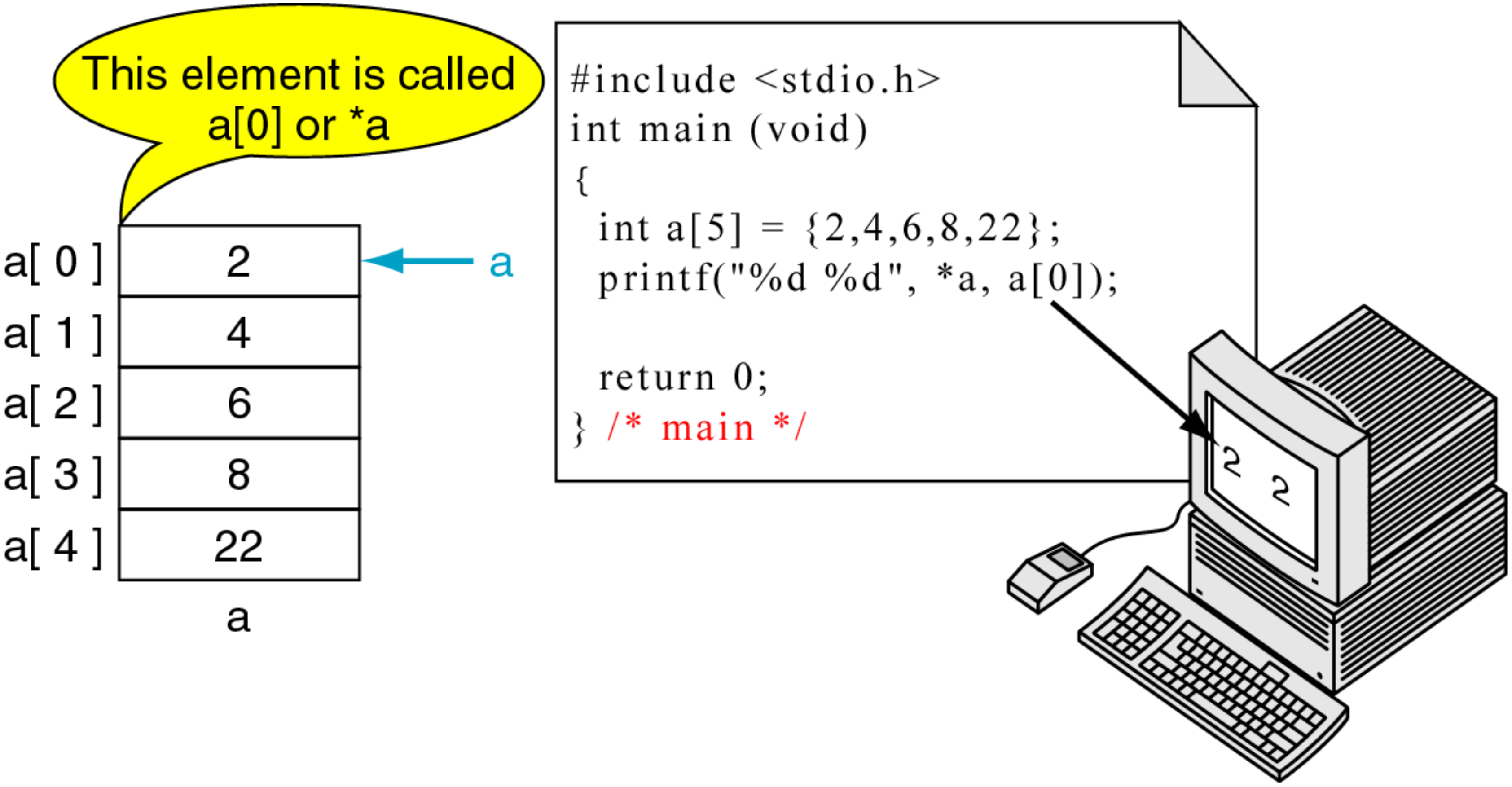
- Note: both values will be same

Dereference of Array name

- Variation – prove that the array name is a pointer constant to the first element of the array

- Program: Demonstrate by printing the value in the first element of the array using both a pointer and an index

Figure 11-2 Dereference of Array name



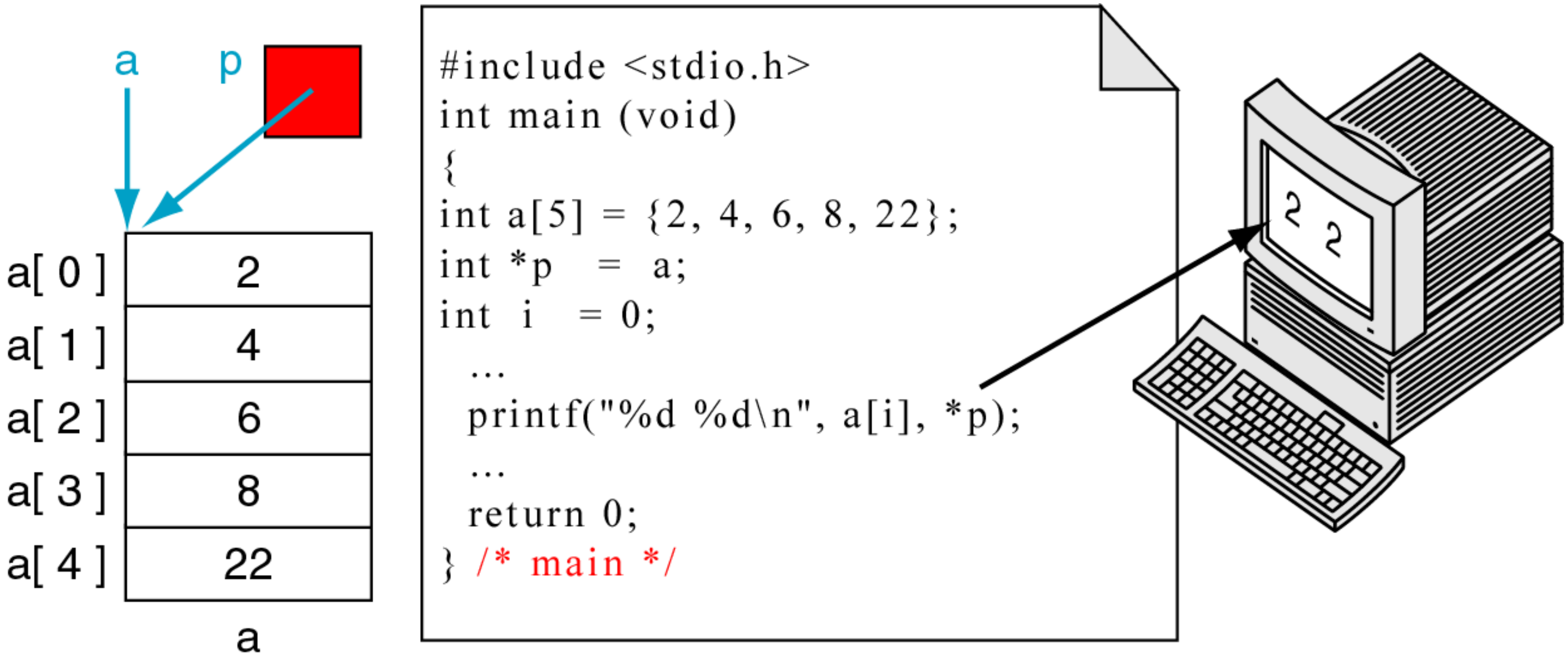
```
#include <stdio.h>
int main (void)
{
  int a[5] = {2,4,6,8,22};
  printf("%d %d", *a, a[0]);

  return 0;
} /* main */
```

Array names as pointers

- If the name of an array is a pointer, show that we can store this pointer in a pointer variable and use it in the same way we use the array name
- Define and initialize the array
- Define a pointer and initialize it to point to the first element of the array by assigning the array name
- Print the first element in the array using both index and pointer notations

- Note: the array name is unqualified. There is no address operator or index

Figure 11-3 Array names as pointers



```
#include <stdio.h>
int main (void)
{
int a[5] = {2, 4, 6, 8, 22};
int *p   =  a;
int  i   = 0;
  …
  printf("%d %d\n", a[i], *p);
  …
  return 0;
} /* main */
```

Array and a pointer

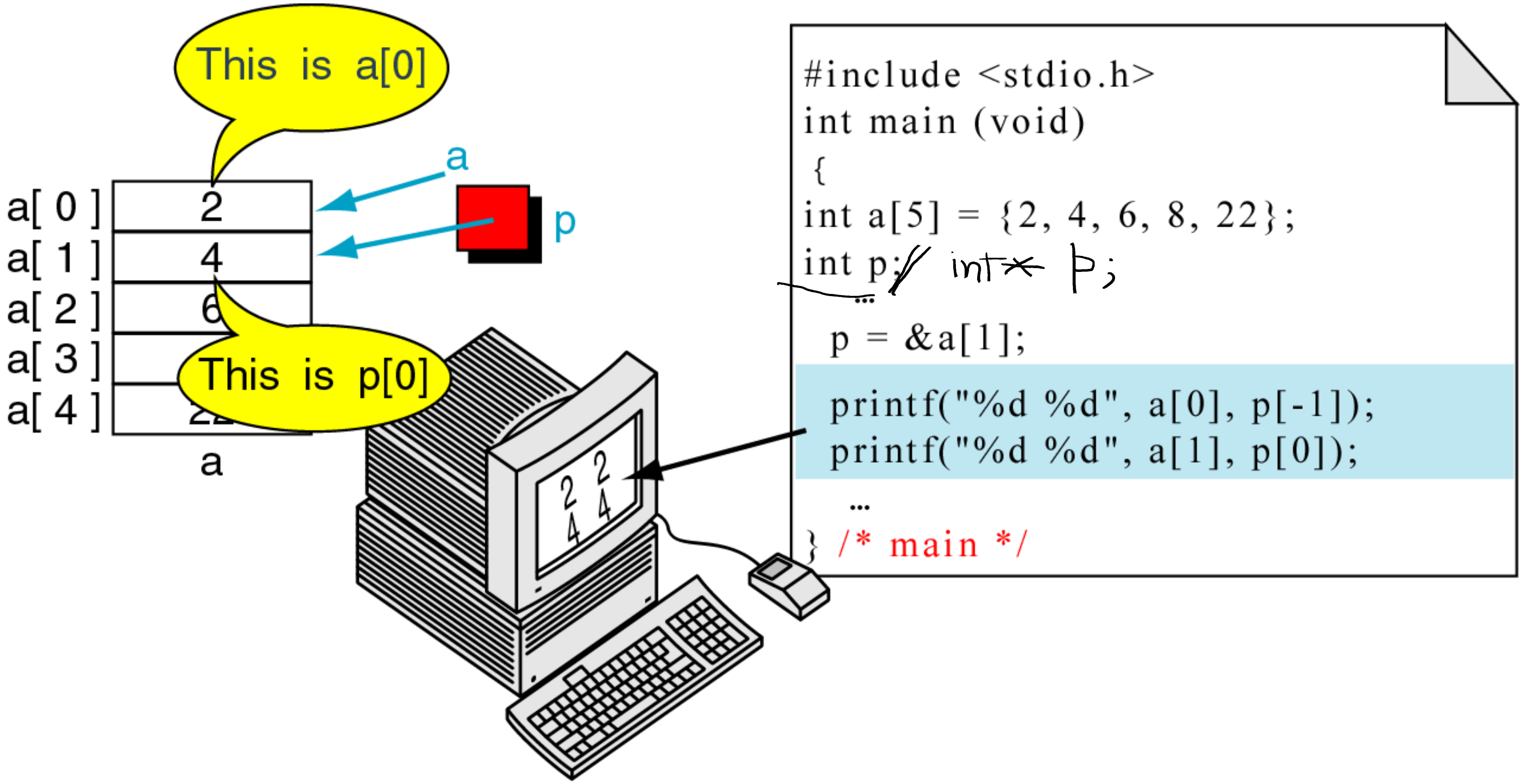
- Program: demonstrate the use of multiple names for an array to reference different locations at the same time

1) by an array name

2) create a pointer to integer and set it to the second element of the array a[1]

- This pointer can be used as an array name and can be indexed
- Print the first 2 elements using array name as well as by pointer
- Note: negative index

Figure 11-4 Multiple Array Pointers



```
#include <stdio.h>
int main (void)
 {
int a[5] = {2, 4, 6, 8, 22};
int p;// int* p;
   ...
  p = &a[1];
  printf("%d %d", a[0], p[-1]);
  printf("%d %d", a[1], p[0]);
   ...
} /* main */
```

- To access an array, any pointer to the first element can be used instead of the name of the array

Pointer Arithmetic and Arrays

Pointer Arithmetic and Arrays

- Moving through an array:
  1. indexing
  2. Pointer Arithmetic
- Pointer Arithmetic: offers a set of arithmetic operators for manipulating the addresses in pointers
- Powerful for moving element by element in an array (search sequentially)

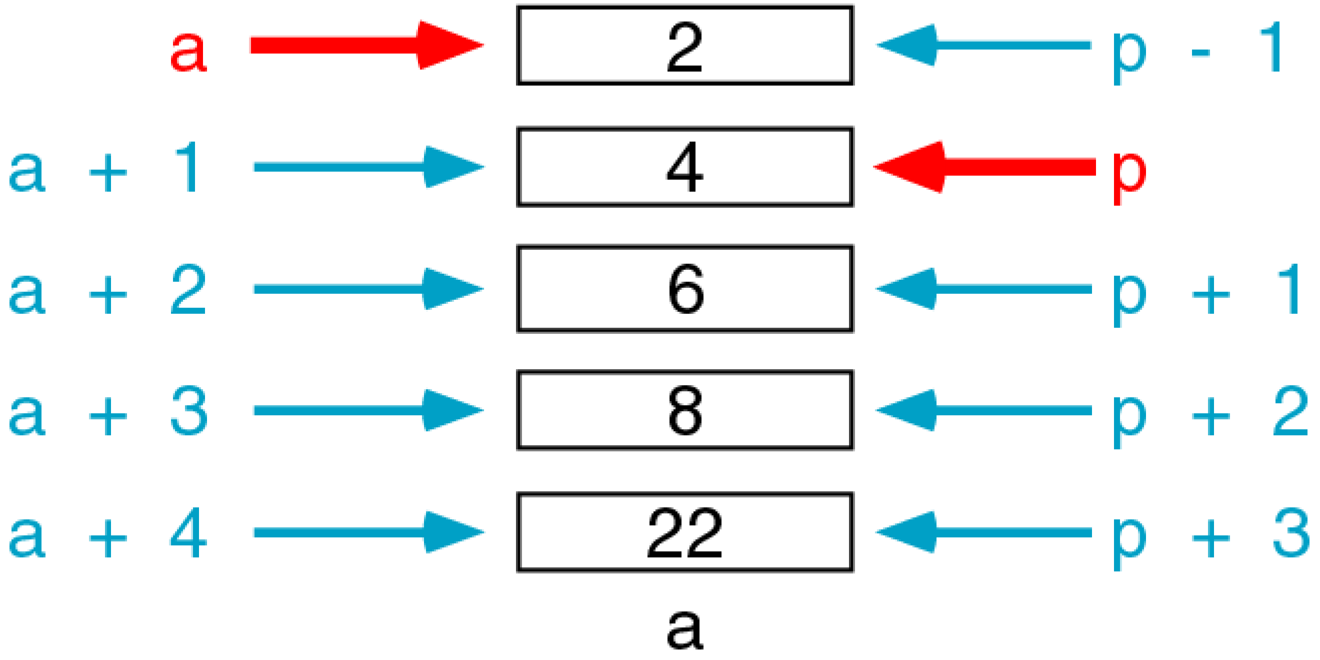
Pointers and One-Dimensional Arrays

- If we have an array, a, then
  - a is a constant pointing to the first element
  - a+1 is a constant to the second element
- If p is a pointer pointing to a[1], then
  - p-1 is a pointer to the previous (first) element
  - p +1 is a pointer to the next (third) element
- Generalizing:
- Given pointer p, p + n or p-n is a pointer to the value n elements away

Pointers and One-Dimensional Arrays

- As long as a or p are pointing to one of the elements of the array
  - we can add or subtract to get the address of other elements of the array

Figure 11-5 Pointer Arithmetic



- Different from normal arithmetic
- Adding an integer n to a pointer value gives a new value that corresponds to an index location, n elements away
- So n is an offset from the original pointer

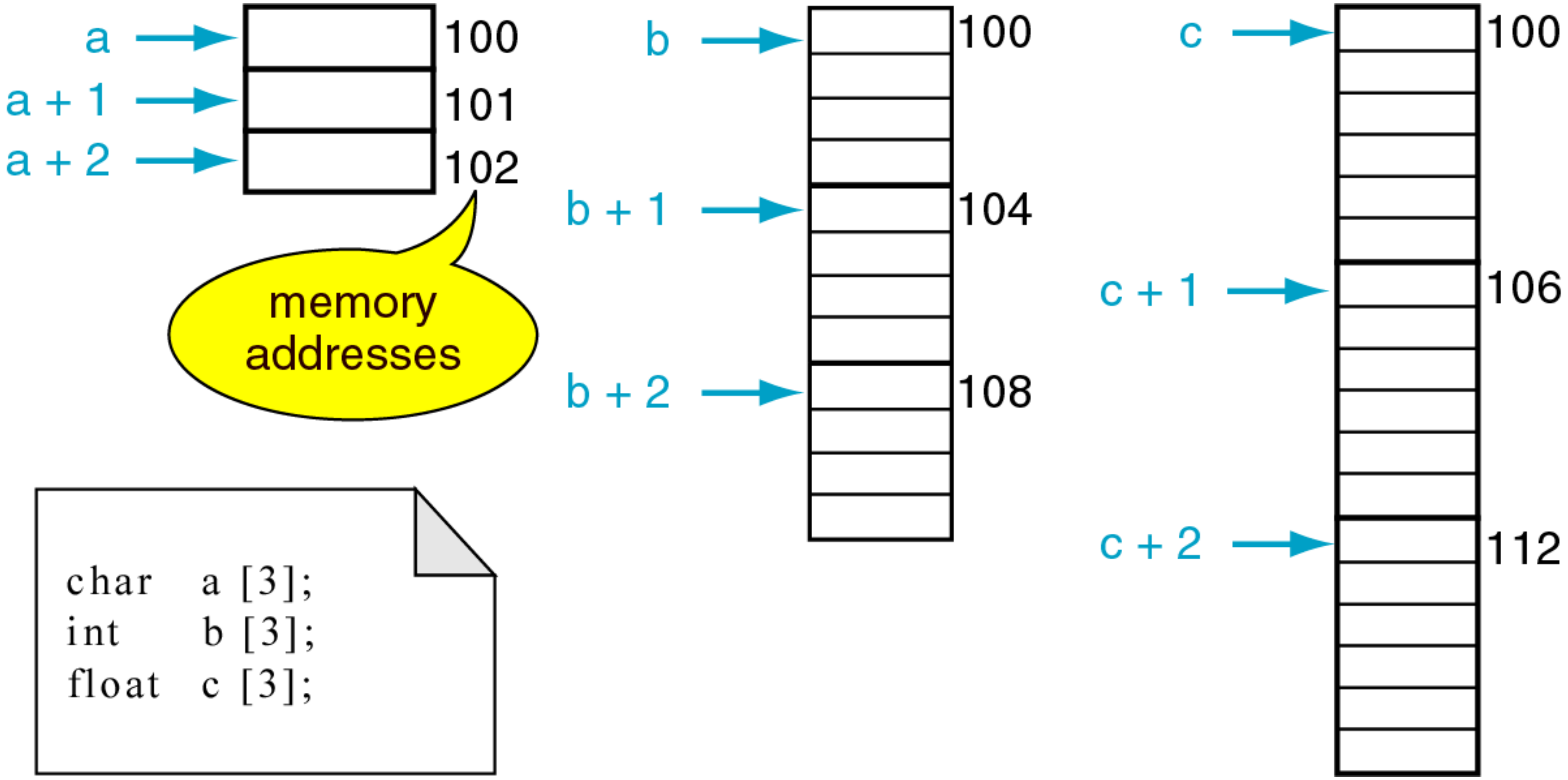
Pointers and One-Dimensional Arrays

- To determine the new value, C must know the size of one element
- The size of element is determined by the type of the pointer
- This is the reason why pointers of different types cannot be assigned to each other

Pointers and One-Dimensional Arrays

- If offset is 1:
  - C will add or subtract one element size from the current pointer value
  - access is more efficient than corresponding index notation
- If offset is more than 1:
  - C must compute the offset by multiplying the offset by the size of one array element and adding it to the pointer value as below:
  - address = pointer + (offset * size of element)
  - Depending on the hardware the multiplication can make it less efficient than simply adding 1
  - So pointer arithmetic is not efficient compared to indexing
  - a + n becomes a + n * (sizeof(one element))

Figure 11-6 Pointer Arithmetic and different types



```
char    a [3];
int     b [3];
float   c [3];
```

- Pointer arithmetic on different sized elements

Pointer arithmetic on different sized elements

- a+1means different things:
- char takes 1 byte:
  - Adding 1 moves to the next memory address (101)
- Integer taking 4 bytes:
  - Adding 1 to array pointer b moves to 4 bytes in memory (104)
- Float taking 6 bytes:
  - Adding 1 to array pointer c moves to 6 bytes in memory (106)

Note: size is compiler dependent

Pointers and One-Dimensional Arrays

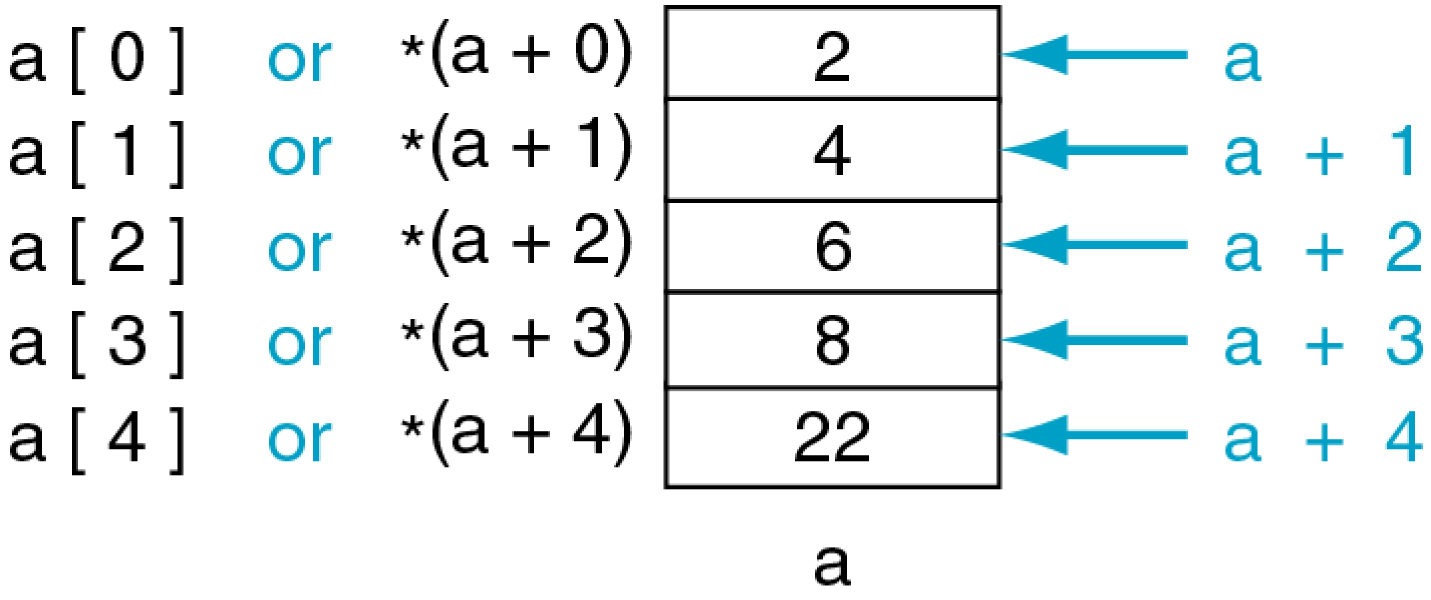
- We have seen how to get the address of an array element using a pointer and an offset
- How can we use that value?

1) we can assign it to another pointer

```
p = arrayName + 5;
```

2) use with indirection operator to access or change the value of the element we are pointing to

Figure 11-7 Dereferencing array pointers



a [0] or *(a + 0) 2 a

a [1] or *(a + 1) 4 a + 1

a [2] or *(a + 2) 6 a + 2

a [3] or *(a + 3) 8 a + 3

a [4] or *(a + 4) 22 a + 4

a

a[n] and *(a+n) are identical expressions

Program to find the smallest number

- Find the smallest number among 5 integers stored in an array
- pSm is set to the first element of the array
- pWalk is working pointer pointing to the second element
- Working pointer advances through the remaining elements
- Compares current element with element pointed to by pSm & assigning smaller to pSm

Figure 11-8 Find smallest

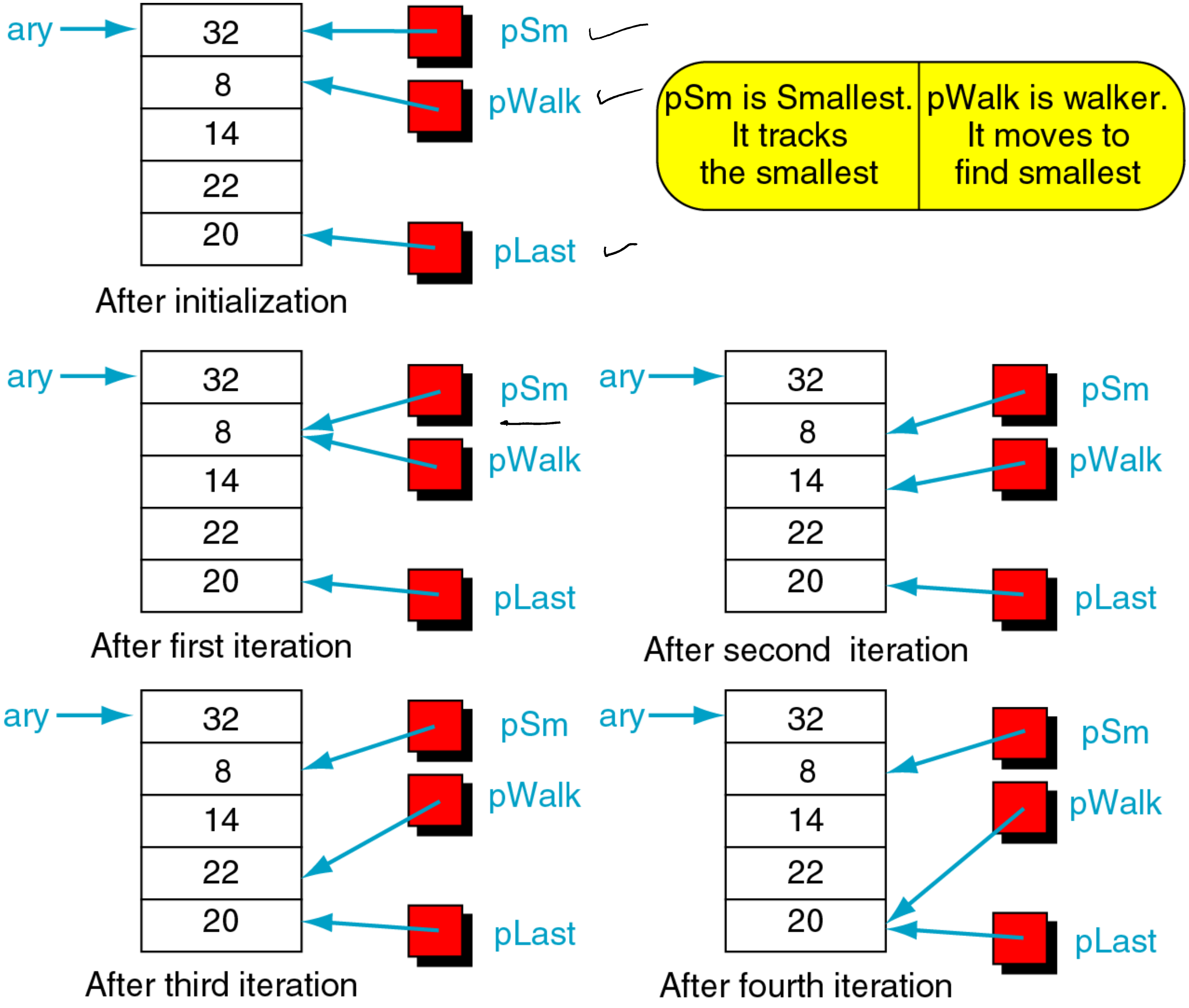
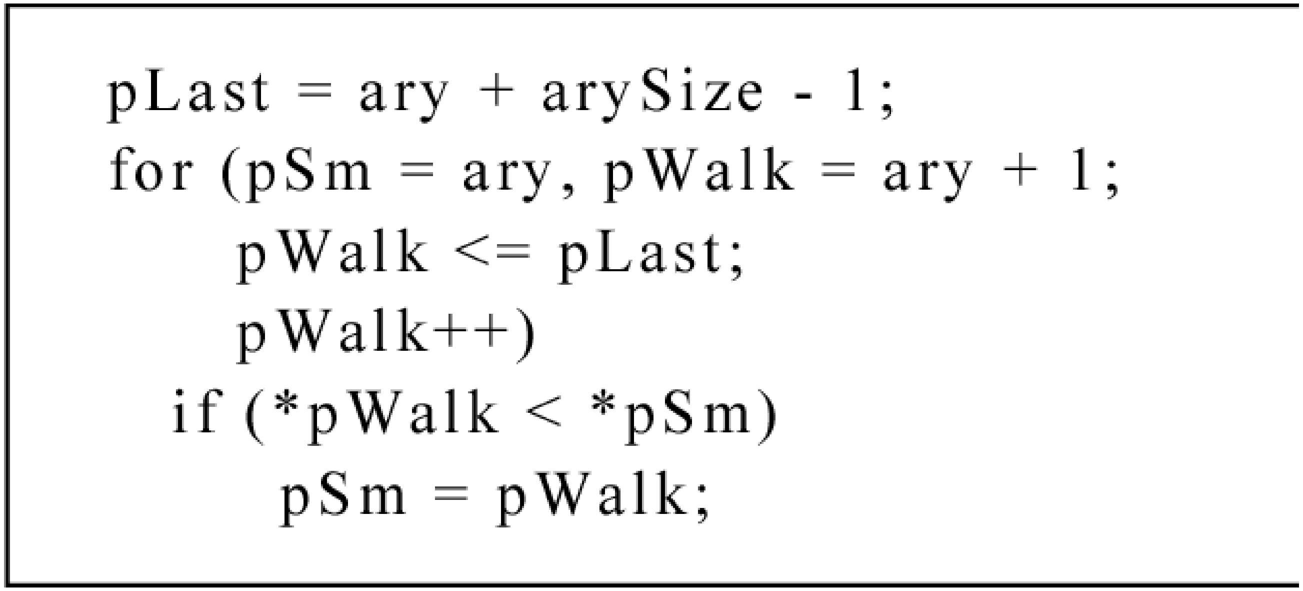


Figure 11-8 Find smallest



```
pLast = ary + arySize - 1;
for (pSm = ary, pWalk = ary + 1;
     pWalk <= pLast;
     pWalk++)
  if (*pWalk < *pSm)
       pSm = pWalk;
```

Arithmetic Operations on Pointers

- Arithmetic operations involving pointers
- Addition can be used when one operand is a pointer and the other is an integer
- Subtraction can be used only when both operands are pointers or when one operand is a pointer and the other is an index integer
- Note: result is meaningful only if the two pointers are associated with the same array structure
- Also, pointer with postfix and unary increment and decrement operators are valid
- Ex: p+5, 5+p, p-5, p1-p2, p++, --p are valid

Pointers and other operators

- Relational operators involving pointers
- Allowed only if both operands are pointers of the same type
- Ex: pointer relational expns
- P1 >= p2 , p1 != p2
- If(ptr != NULL) same as if(ptr)
- If (ptr == NULL) same as if(!ptr)

Using Pointer Arithmetic

Using pointer arithmetic

- Program: demonstrate moving through array using pointers forward and backward

Program 11-1 Print Array with Pointers

```
#define MAX_SIZE 10
int main(void){
    int ary[] ={1,2, -- , 10};
    int * pWalk;
    int * pEnd;
    //print array forward
    for(pWalk= ary, pEnd= ary+MAX_SIZE;
            pWalk < pEnd ; pWalk ++)
          printf ("%3d", *pWalk);
   printf("\n");
   //print array backward
   for (pWalk=pEnd-1; pWalk >=ary; pWalk--)
         printf("%3d", *pWalk);
   printf ("\n");
   return 0;

}
```

Binary search

- Find 22 in a sorted array: 12 is the size of list
- List: { 4, 7, 8, 10, 14, 21, 22, 36, 62, 77, 81, 91}

Pointers & the Binary search

list must contain at least one element

endPtr is pointer to largest element in the list

target is value of element being sought

If Found

locnPtr points to target element

```
int binarySearch(int list[], int* endPtr, int target, int** locnPtr) {
  int* firstPtr, midPtr, lastPtr;//Local Declarations
  firstPtr = list;
  lastPtr = endPtr;
  while (firstPtr <= lastPtr) {
        midPtr = firstPtr + (lastPtr - firstPtr) / 2;
        if (target > *midPtr)
            firstPtr = midPtr + 1;
        else if (target < *midPtr )
                lastPtr = midPtr - 1;
            else
                firstPtr = lastPtr + 1;
        }
    *locnPtr = midPtr;
    return (target == *midPtr);
}
```

Pointers and 2D arrays

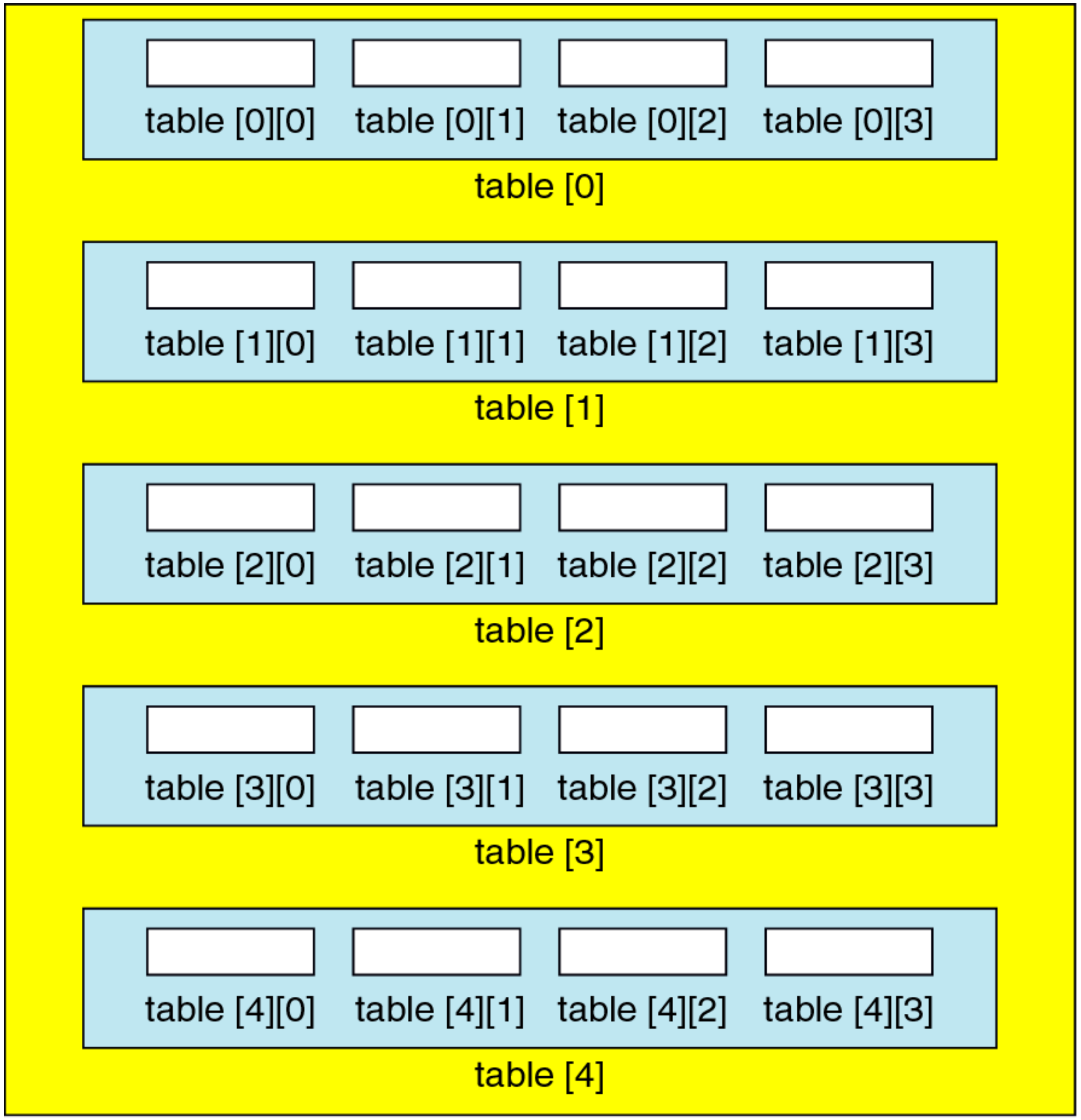


table [0][0] table [0][1] table [0][2] table [0][3]

table [0]

table [1][0] table [1][1] table [1][2] table [1][3]

table [1]

table [2][0] table [2][1] table [2][2] table [2][3]

table [2]

table [3][0] table [3][1] table [3][2] table [3][3]

table [3]

table [4][0] table [4][1] table [4][2] table [4][3]

table [4]

table

Pointers to 2D arrays

- Just as in 1D array, the name of the array is a pointer constant to the first element of the array
- Here, the first element is another array
- Suppose, we have an array of integers
- For a 2D array, when we dereference the array name, we get an array of integers (we do not get one integer)
- So, dereferencing of the array name of a 2D array is a pointer to a 1D array

Figure 10-9 Pointers to 2D arrays

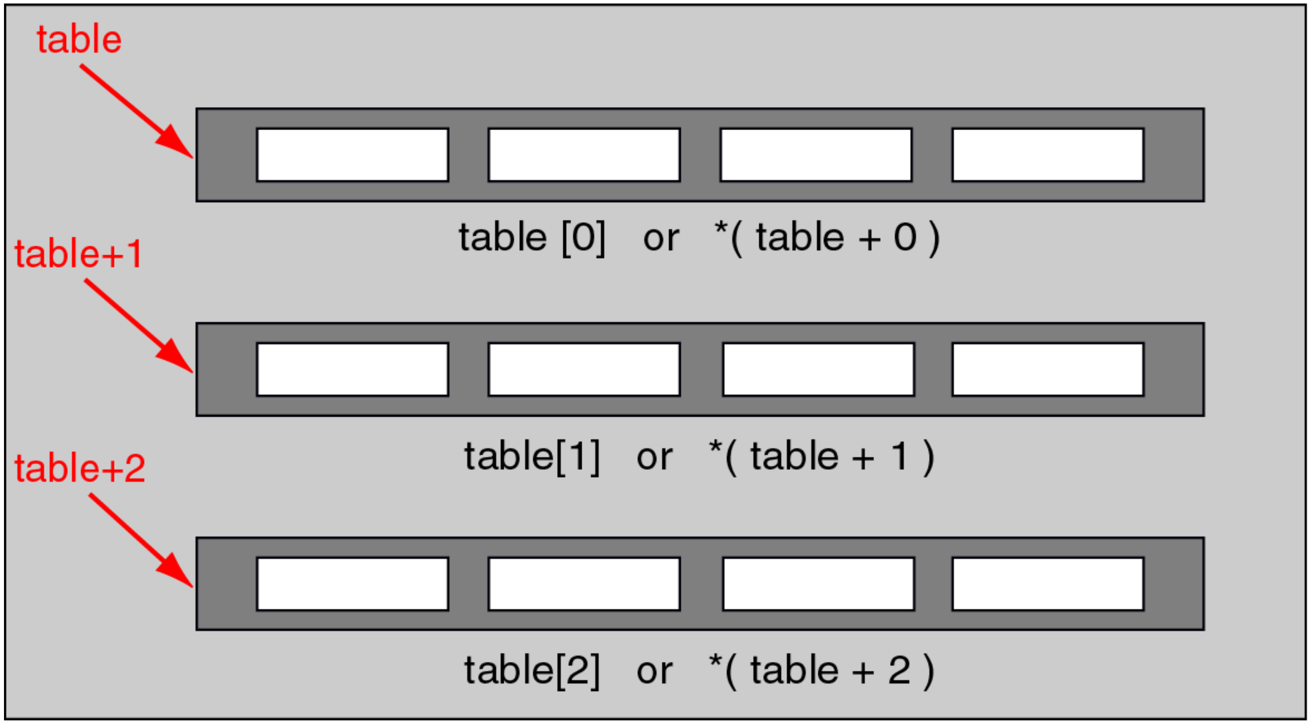
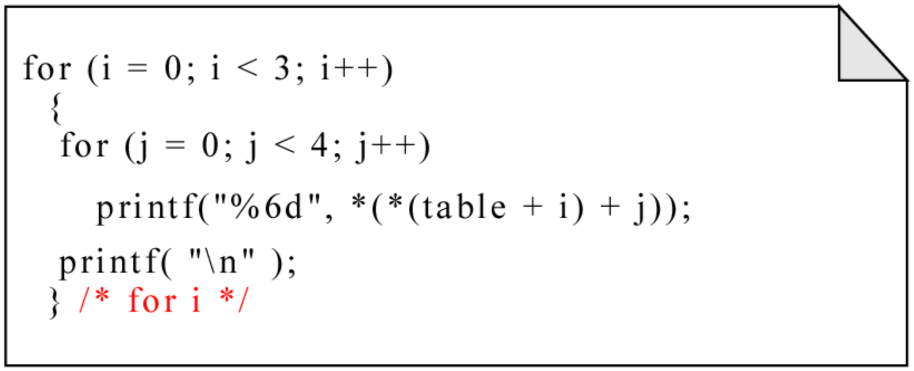


table
table+1
table+2
table [0]   or   *(table + 0)
table[1]   or   *(table + 1)
table[2]   or   *(table + 2)

int table[3][4] ;



```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)

      printf("%6d", *(*(table + i) + j));

   printf( "\n" );
  } /* for i */
```

Print table

Pointers to 2D arrays

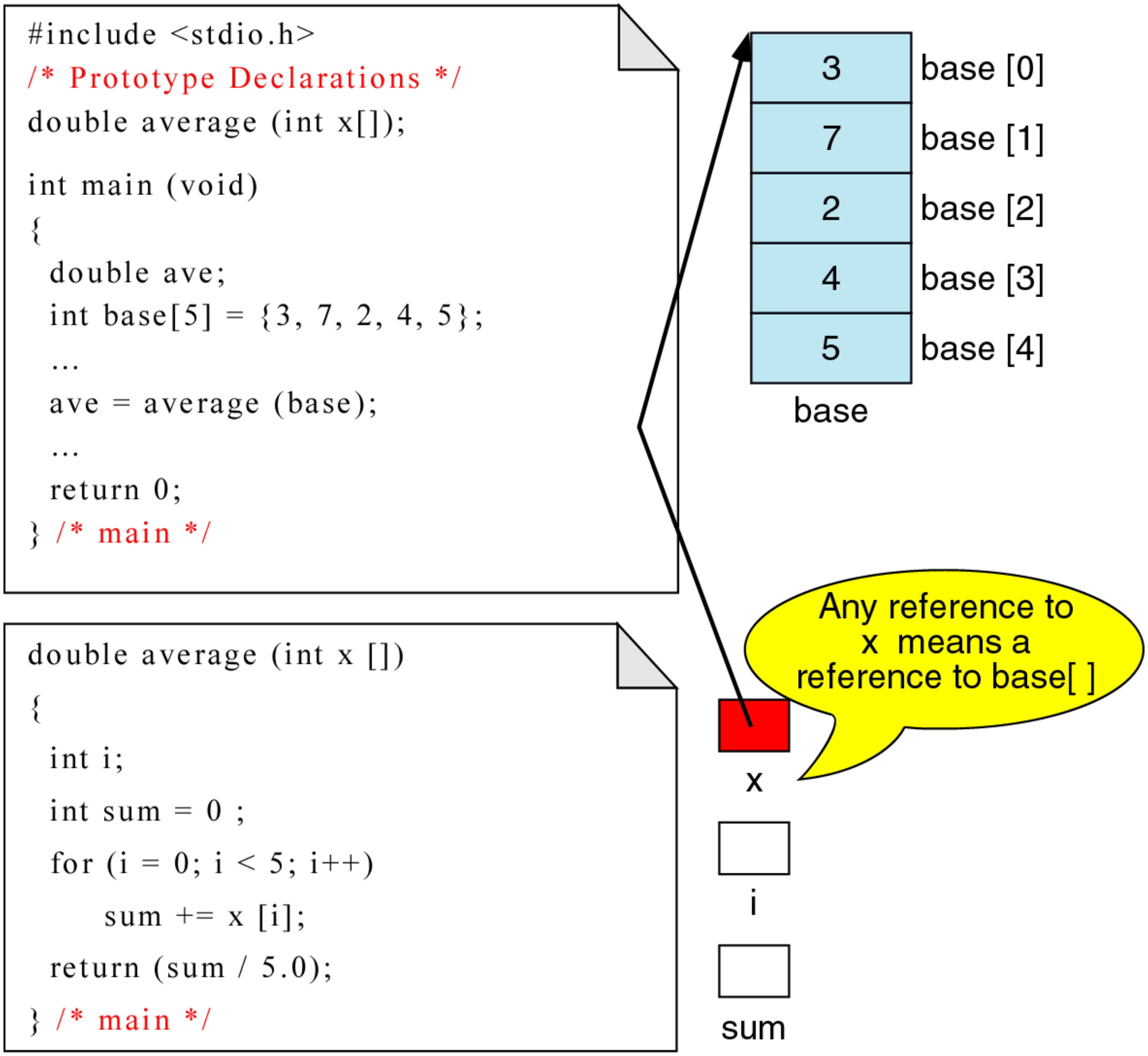
- In the figure, each element is shown in both index and pointer notation
  - table[0]: refers to an array of 4 integer values
  - *(table+0) : equivalent pointer notation is by dereferencing the array name
- Program: demonstrate pointer manipulation with 2D array by printing the table
  - To refer to a row: dereference the array pointer which gives us a pointer to a row
  - Given a pointer to a row, to refer to an element, dereference the row pointer
  - So it leads to double dereference as *(*(table))

Pointers to 2D arrays

- The double dereference *(*(table)) : refers only to the first element of the first row
- To step through all the elements, we need 2 offsets:
  - One for the row
  - Another for the element within the row
- i and j: loop counters for offsets
- Same logic as printing 2D array using indexes
  - table [i][j]: index notation
  - *(*(table+i)+j) : pointer notation
- Note: With multi-dimensional arrays, the pointer arithmetic is not efficient over indexing
- As it is complex, index notation is preferred

Passing an array to a function

Figure 8-11 passing arrays



```
#include <stdio.h>
/* Prototype Declarations */
double average (int x[]);

int main (void)
{
  double ave;
  int base[5] = {3, 7, 2, 4, 5};
  ...
  ave = average (base);
  ...
  return 0;
} /* main */
```

```
double average (int x [])
{
  int i;
  int sum = 0 ;
  for (i = 0; i < 5; i++)
       sum += x [i];
  return (sum / 5.0);
} /* main */
```

Passing an array to a function

- If array name is a pointer to the first element, we can pass the array name to a function
- When we pass the array name, no need to use the address operator
- The array name is a pointer constant, so the array name is already the address of the first element in the array
- Ex: compute(list)   // calling program
- 1) int compute(int ary[])

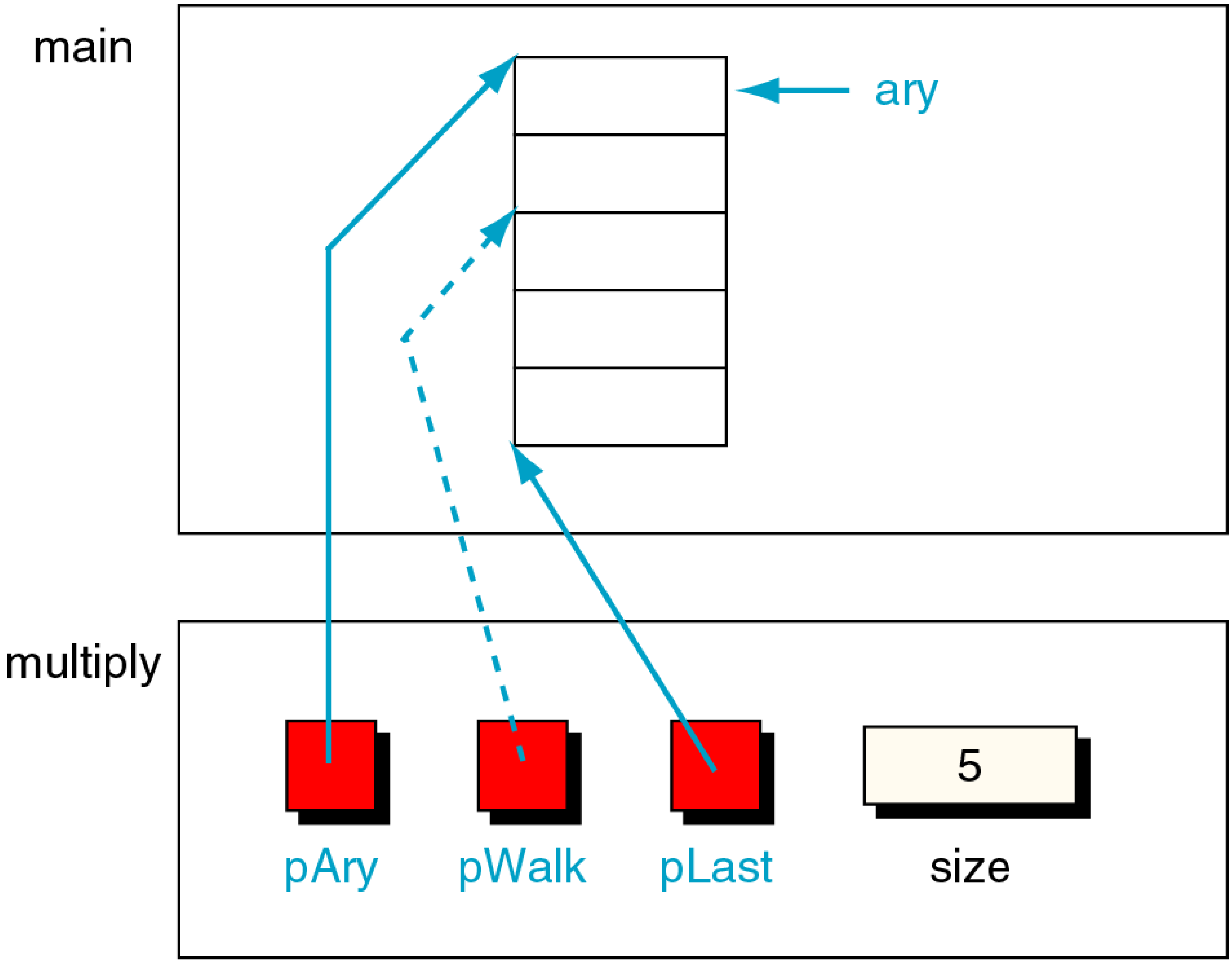
Passing an array to a function

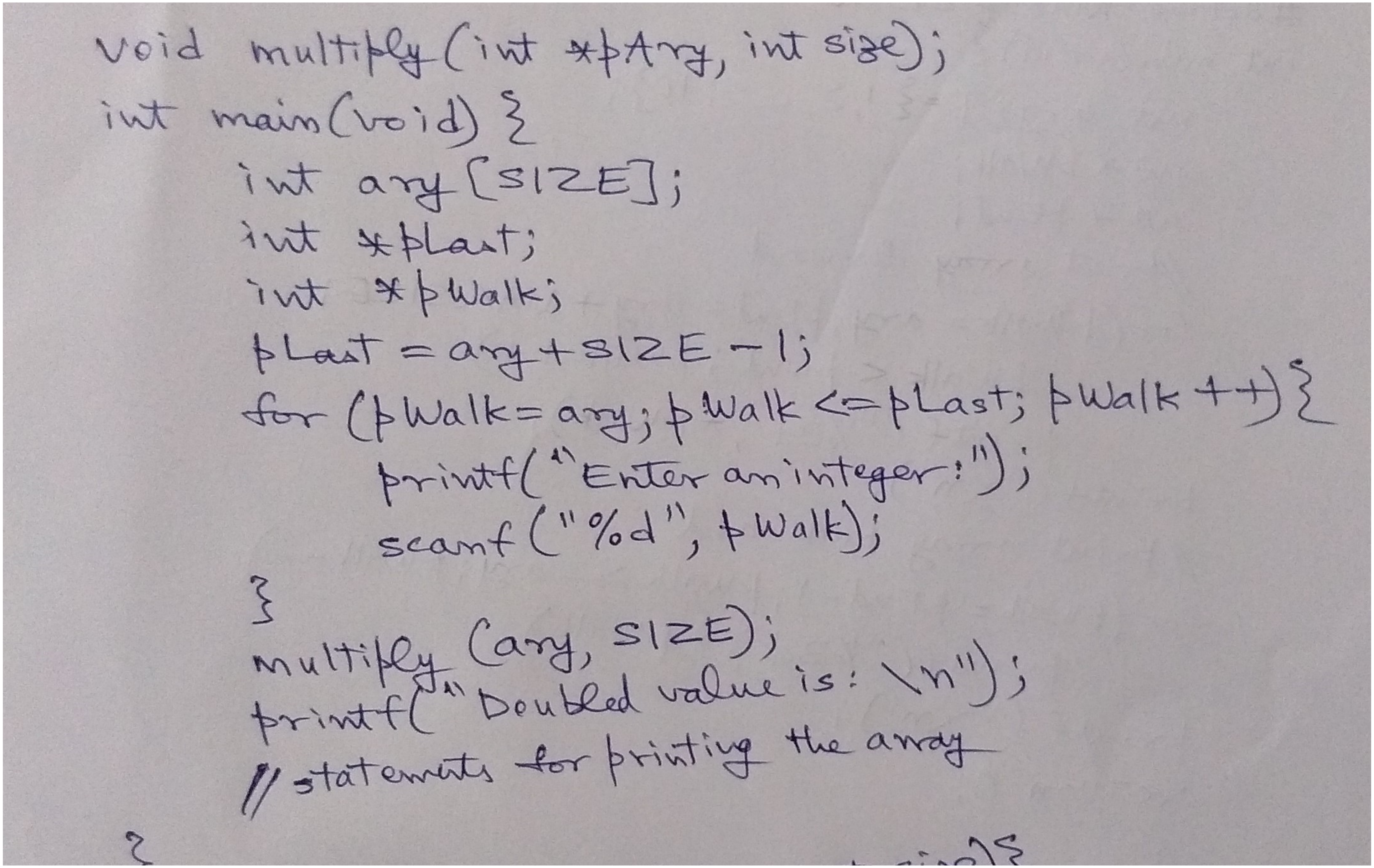
- 2) int compute(int *arySalary)   //needs documentation that array is passed
- 3) for a 3D array int compute(int bigary[][2][5])   //the compiler needs to know the size of dimension after the first to calculate the offset for pointer arithmetic

Passing an array to a function

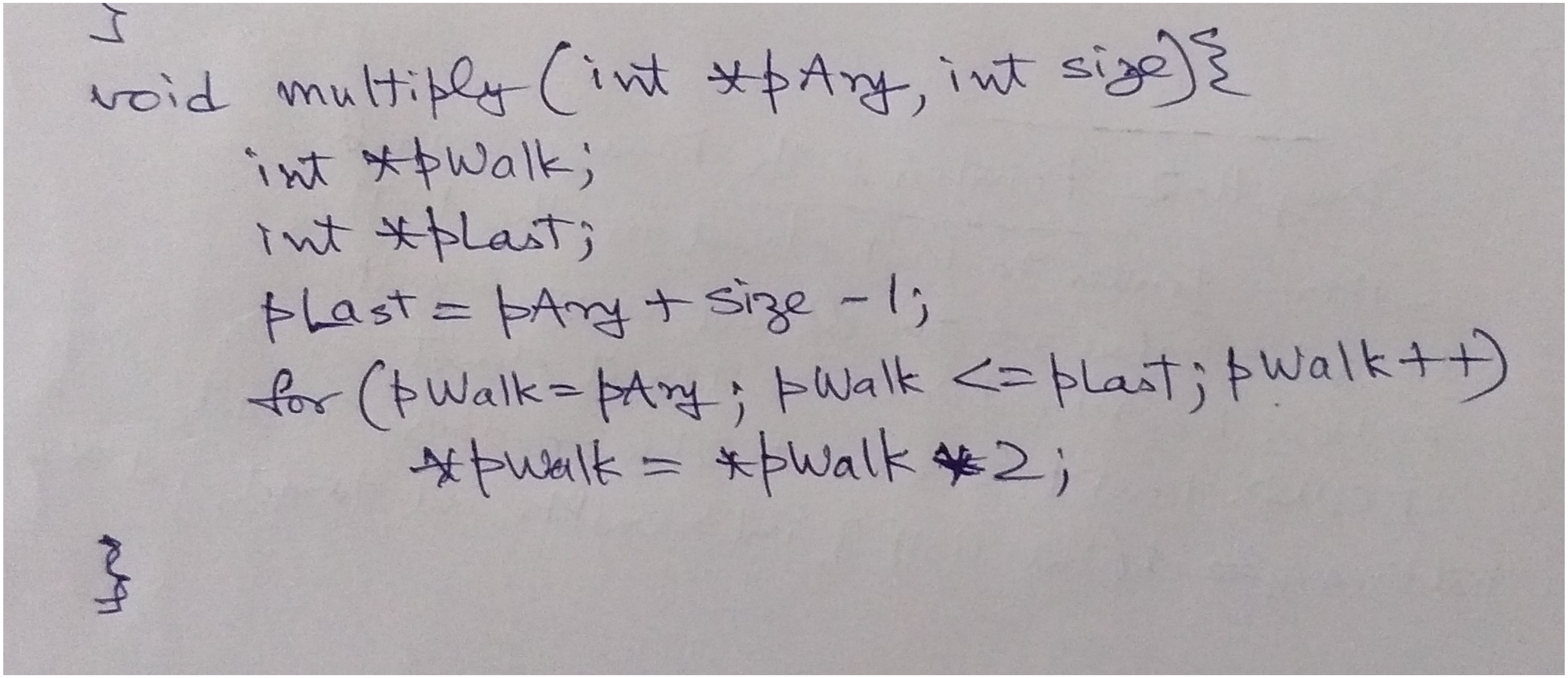
- Program: call a function to multiply each element of a 1D array by 2
- The variables are shown next

Figure 10-10 Variables for multiply array elements by 2





```
void multiply(int *pAry, int size);
int main(void) {
        int ary[SIZE];
        int *pLast;
        int *pWalk;
        pLast = ary+SIZE-1;
        for (pWalk=ary; pWalk <=pLast; pWalk++){
              printf("Enter an integer:");
              scanf("%d", pWalk);
        }
        multiply (ary, SIZE);
        printf("Doubled value is: \n");
        // statements for printing the array
}
```



```
}
void multiply(int *pAry, int size){
        int *pWalk;
        int *pLast;
        pLast = pAry + size - 1;
        for(pWalk=pAry; pWalk <=pLast; pWalk++)
                *pWalk = *pWalk *2;

}
```

Memory Allocation Functions

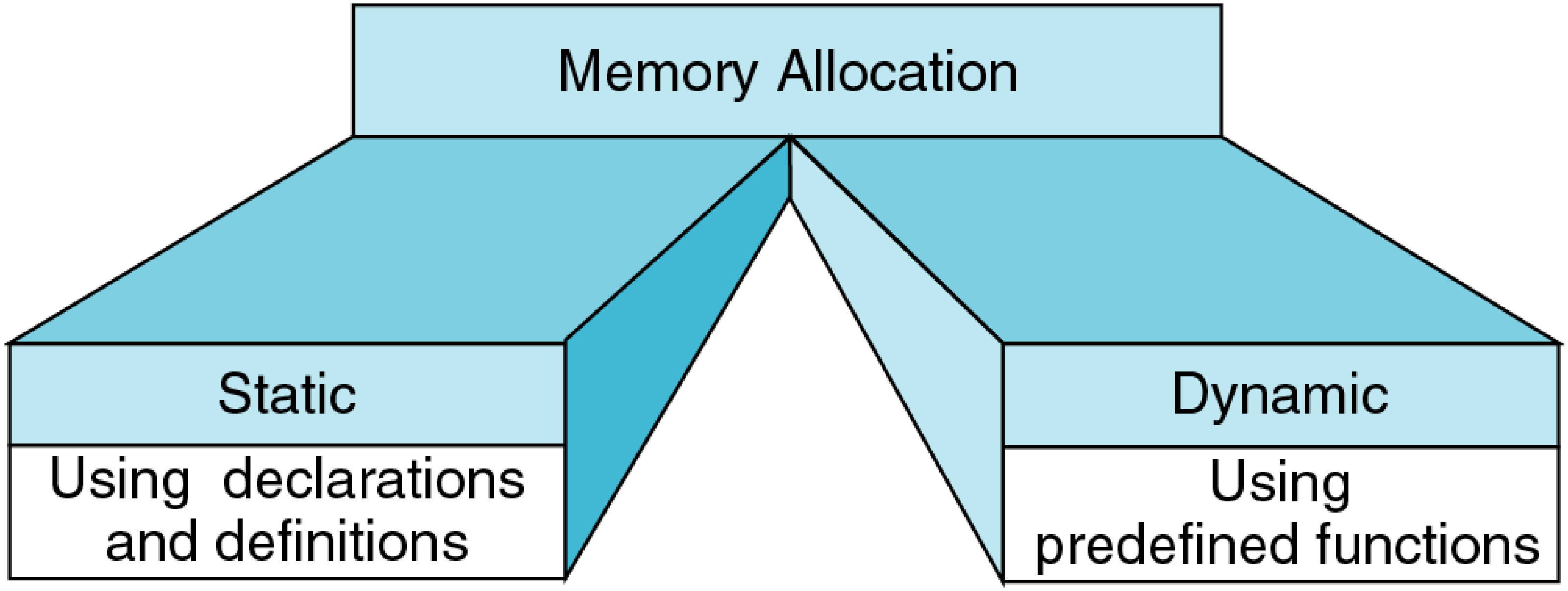
Memory Allocation Functions

- Two choices to reserve memory locations for an object:
- 1. Static memory allocation
- Requires that the declaration and definitions of memory be fully specified in the source program
- The number of bytes reserved cannot be changed during run time
- Works fine if the data requirements are exactly known
- This is what we have done so far
- Note: earlier PLs (FORTRAN, COBOL) had this limitation. The data structure need too be fully specified during compile time

Memory Allocation Functions

- 2. Dynamic memory allocation
- Uses predefined functions to allocate and release memory for data while the program is running
- Postpones the data definition to run time
- Can be used either with standard data types or with derived data types

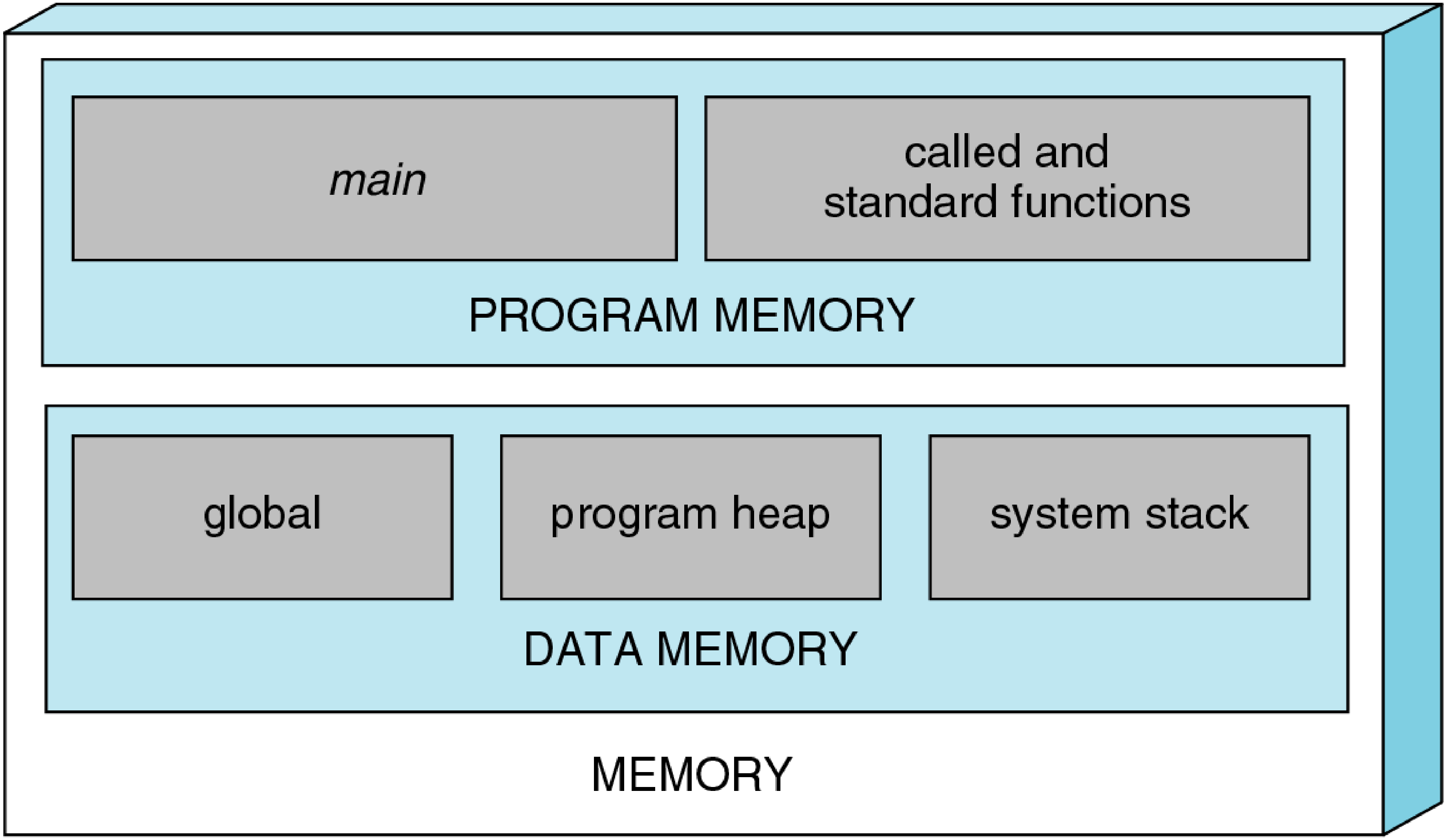
Figure 10-13 Memory Allocation


Memory Allocation
Static
Using declarations and definitions
Dynamic
Using predefined functions

Understanding dynamic Memory Allocation

- Conceptually memory is divided into program memory and data memory

Figure 10-15 A conceptual View of memory



- Note: implementation of memory is left to the designers who design the syste
- Stack and heap can share the same pool of memory
- So the above diagram is only conceptual

Understanding dynamic Memory Allocation

- program memory:
- Consists of the memory used for main and all called functions
- data memory:
- Consists of permanent definitions such as global data and constants, local definitions and dynamic data memory

- Note: Exact ways of handling these needs will be decided by the OS and compiler

Understanding dynamic Memory Allocation

- program memory:
- Consists of the memory used for main and all called functions
- main() must be in memory all the time
- Each called function must be in memory while it is active

- Note: any function has to be in memory when it is running

Understanding dynamic Memory Allocation

- Memory allocation : stack
- Although the program code for a function may be in memory at all times
  - The local variables for the function are available only when it is active
- In recursion, more than one version of the function can be active at a time
  - Hence multiple copies of the local variables are allocated although only one copy of the function is present
- The memory facilities for these capabilities is known as stack

Understanding dynamic Memory Allocation

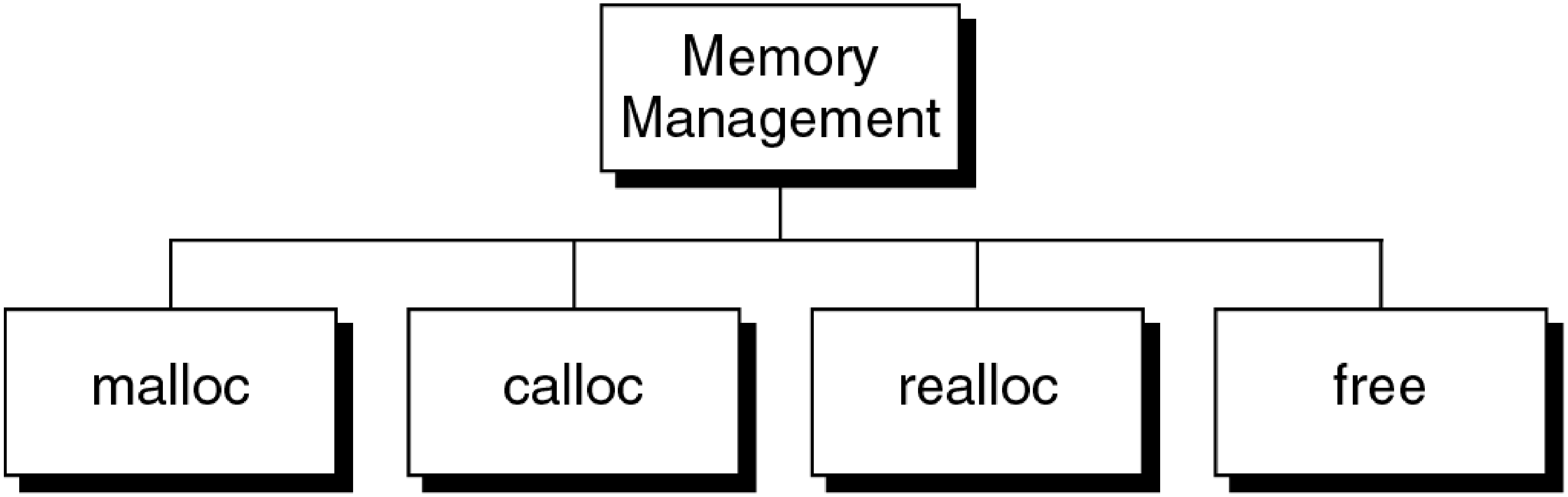
- Memory allocation : heap
- Heap is unused memory allocated to the program
- available to be assigned during its execution
- The memory pool from which memory is allocated when requested by the memory allocation functions

Understanding dynamic Memory Allocation

```
//static memory
int x;

//dynamic memory
int* x;
x = malloc (. . . );
```

```
//static memory
int ary[3];


//dynamic memory
int* ary;
ary = calloc (. . . );
```

Figure 10-14 Memory Management Functions



Note: all memory management functions are defined in <stdlib.h>

Block memory allocation (malloc)

- Allocates a block of memory that contains the number of bytes specified as the parameter
- Returns a void pointer to the first byte of the allocated memory
- void* malloc (size_t size);
- The allocated memory is uninitialized (contains garbage)
- To allocate an integer in the heap
  ```
  pInt = (int*) malloc (sizeof (int));
  ```
- The pointer returned by malloc is shown to be casted as an integer
- For portability use sizeof()

```
//pInt = malloc(4);
```

malloc

- Successful call
- malloc returns the address of the first byte in the memory allocated
- Unsuccessful call
- malloc returns a NULL pointer
- Overflow – attempt to allocate memory from the heap when memory is insufficient

- Note: Refer to the memory allocated in the heap only through a pointer
- It does not have its identifier

calloc (contiguous memory allocation)

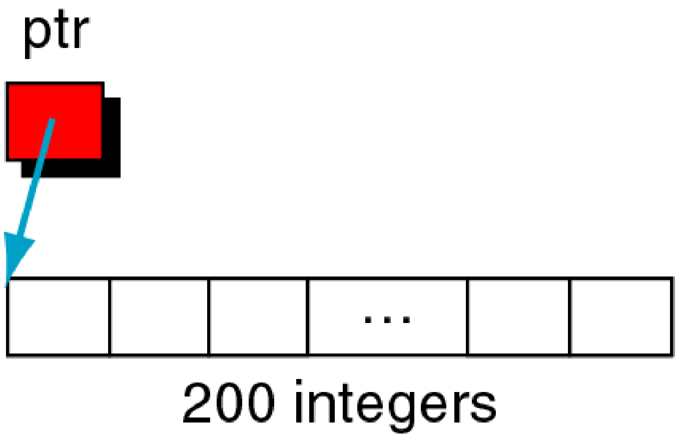
- To allocate memory for arrays
- 1. allocates a contiguous block of memory to contain an array of elements of a specified size
- Parameters – number of elements to be allocated, size of each element
- 2. Returns a pointer to the first element of the allocated array
- Since it is a pointer to an array, the size associated with its pointer is the size of one element not the entire array

calloc (contiguous memory allocation)

- 3. calloc clears memory. So allocated memory is set to zero
- The result is same for malloc and calloc for overflow and zero size
- void* calloc(size_t element_count,

  size_t element_size);

Figure 10-17 calloc



```
if (!(ptr = (int *)calloc (200, sizeof(int))))
    /* No memory available */
    exit (100) ;

/* Memory available */
...
```

ptr = (int*) calloc . . . .

if(!ptr)

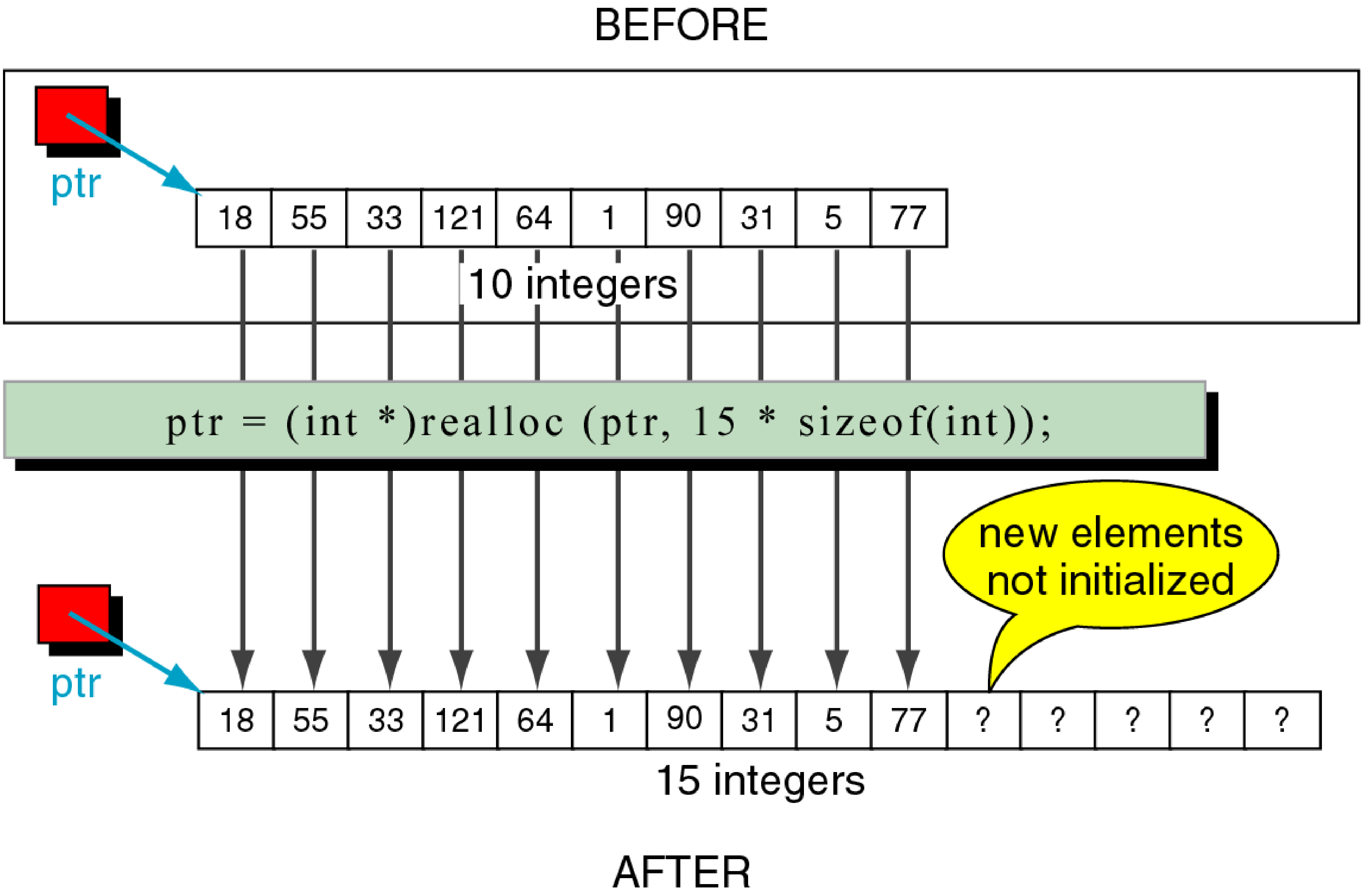
- - -

- Allocating memory for an array of 200 integers

realloc (reallocation of memory)

- Highly inefficient and needs to be used carefully
- Given a pointer to a previously allocated block of memory,
  - realloc changes the size of the block by deleting or extending the memory at the end of the block
  - If the memory cannot be extended (due to other allocations), realloc allocates a completely new block
  - Copies the existing memory allocation to the new allocation and deletes the old allocation
- The programmer must ensure that other pointers to the data are correctly changed
- void* realloc (void* ptr, size_t newSize);

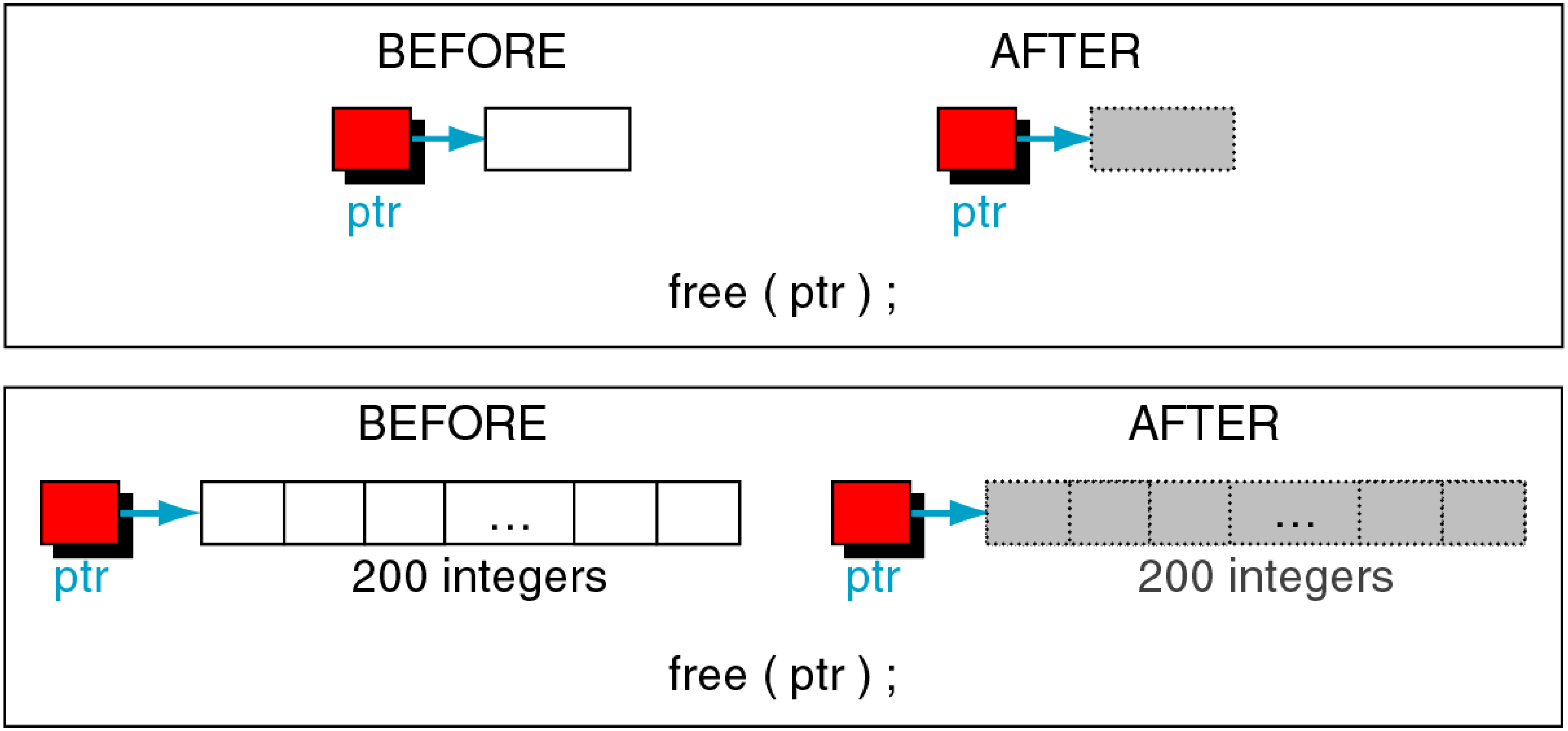
Figure 10-18 realloc



free (releasing memory)

- When memory allocated by malloc, calloc, realloc are no longer needed
  - They can be freed using free
- It is an error to free memory
  - With a null pointer or
  - A pointer to other than the first element of an allocated block
  - A pointer that is a different type than the pointer that allocated the memory
  - To refer to memory after it has been released

Figure 10-19 Freeing memory examples



- Ex1: to release a single element allocated with a malloc back to the heap
- Ex2: : to release a 200 element array allocated with a calloc, all 200 elements are returned back to the heap

free (releasing memory)

- It is not the pointers that are being released but what they point to
- To release an array of memory that was allocated by calloc, release the pointer only once
  - Releasing memory does not change the value in a pointer
  - It still contains the address in the heap
  - It is a logical error to use the memory once it is released
  - After freeing the memory, clear the pointer by setting to NULL (to ease debugging)

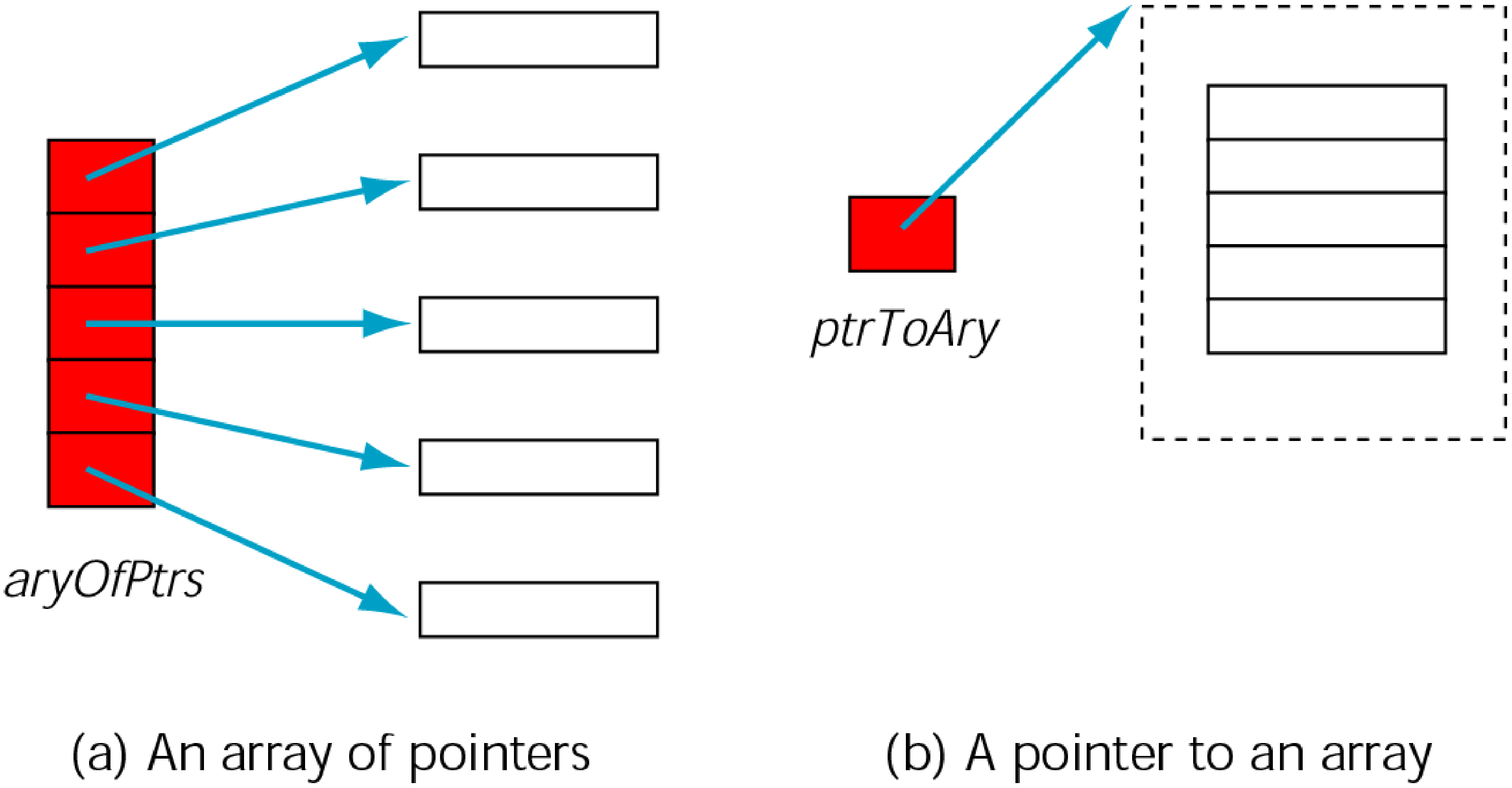
free (releasing memory)

- It is not necessary to clear memory at the end of the program
- OS will release all memory when the program is terminated
- So, free memory whenever it is no longer needed
- void free (void* ptr);

Array of Pointers

- Useful data structure
- Needed when the size of the data in the array is variable
- Ragged arrays
- 2D arrays with an uneven right border
- The rows contain different number of elements (size zero to max)
- 2d array wastes a lot of memory for this structure
- So, create n number of 1D arrays that are joined through array of pointers

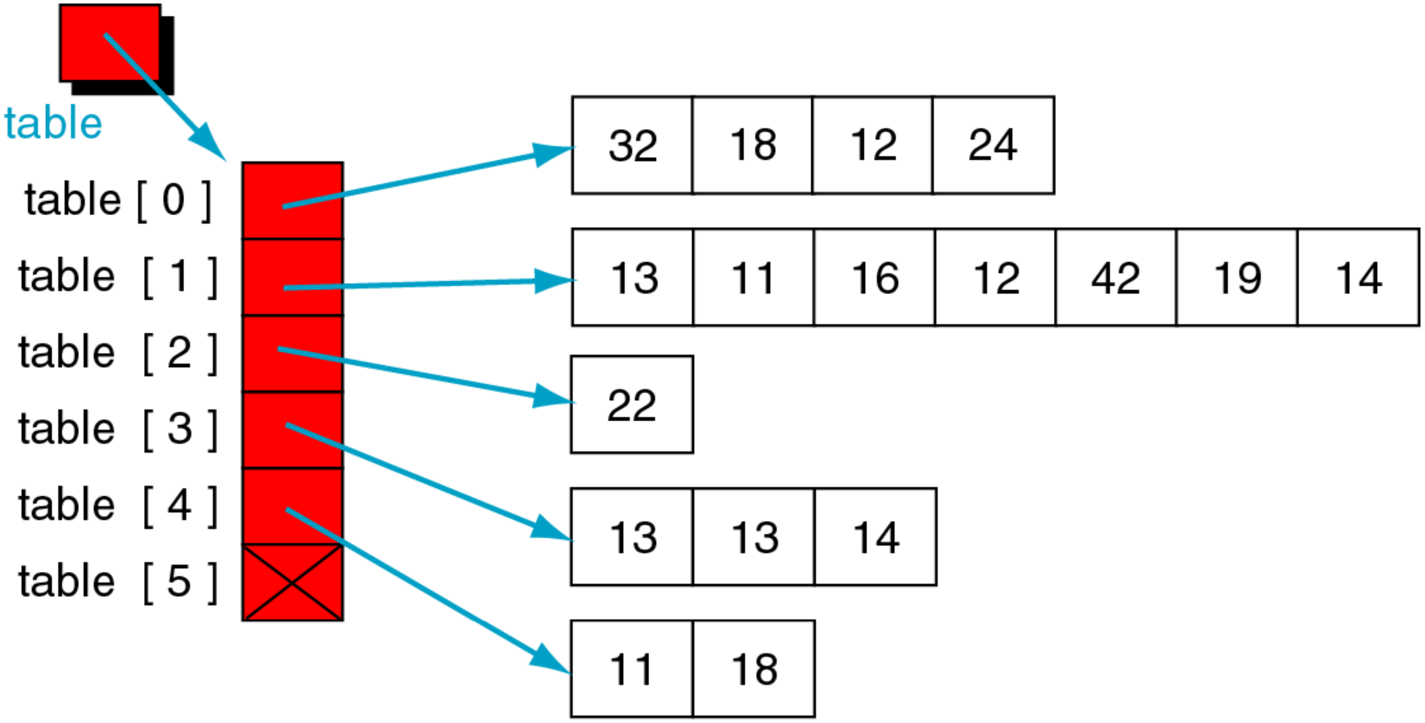
Figure 10-12 Array of pointers vs. Pointer to Array



(a) An array of pointers

(b) A pointer to an array

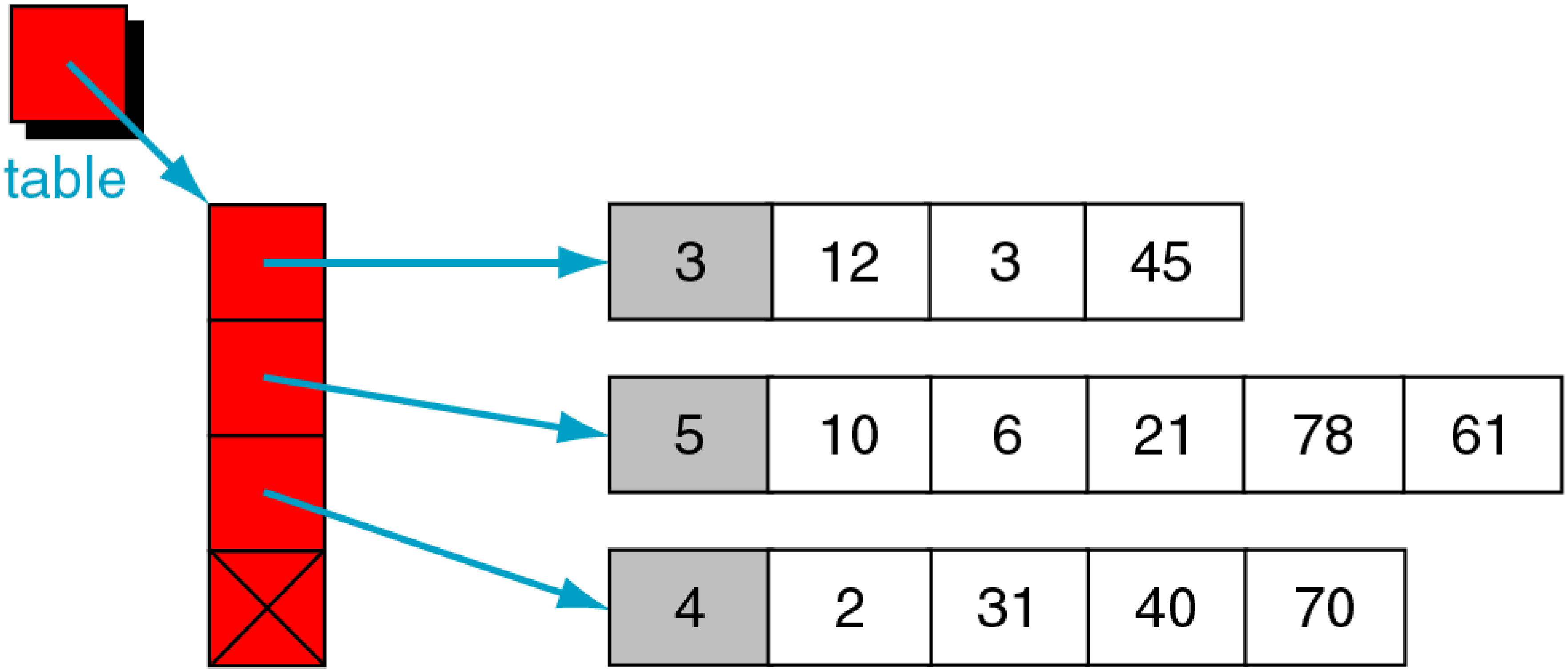
Figure 10-20 A Ragged Array



```
int ** table;
```

```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

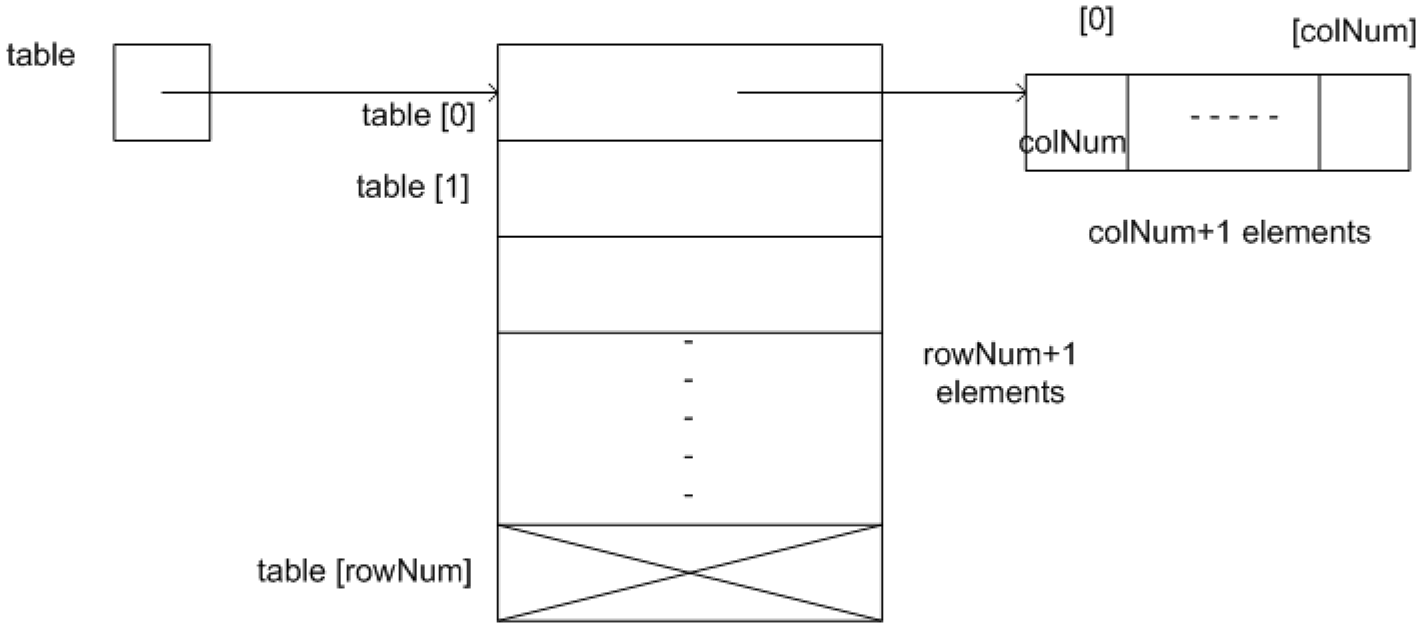
Figure 10-23 ragged array structure



Lab Problem: Construct a ragged array

- The *table* pointer points to the first pointer in an array of pointers.
- Each array pointer points to a second array of integers, the first element of which is the number of elements in the list.
- A sample ragged array structure is shown next

Lab Problem: Construct a ragged array



Lab Problem: Construct a ragged array

- step 1: Declare a ragged array as a variable *table*.
- step 2: Ask the user for row size and set a variable – *rowNum*
- step 3: Allocate space for *(rowNum+1)* pointers as row pointers. The last row pointer will hold NULL
- step 4: Ask the user for column size and set a variable – *colNum*

Lab Problem: Construct a ragged array

- step 5: Allocate space for *(colNum+1)* data elements. The first element will hold value contained in colNum itself.
- step 6: Repeat step 3 for all rows
- step 7 : Display ragged array contents.

Lab Problem: Construct a ragged array

```
# include<stdio.h>
# include<stdlib.h>
int main(){
int rowNum, colNum, i, j;
int **table;
printf("\n enter the number of rows \n");
scanf("%d", &rowNum);
table = (int **) calloc(rowNum+1, sizeof(int *));
for (i = 0; i < rowNum; i++)  /* this will tell which row we are in */
{
        printf("enter size of %d row", i+1);
```

Lab Problem: Construct a ragged array

```
scanf("%d", &colNum);
      table[i] = (int *) calloc(colNum+1, sizeof(int));
          printf("\n enter %d row elements ", i+1);
                  for (j = 1; j <= colNum;  j++)
                  {
                    scanf("%d", &table[i][j]);
                   }
      table[i][0] = colNum;
      printf("size of row number [%d] = %d", i+1, table[i][0]); }
```

Lab Problem: Construct a ragged array

```
table[i] = NULL;
for (i = 0; i < rowNum; i++) /* this will tell which row we are in */
{
      printf("displaying %d row elements\n", i+1);
            for (j = 0; j <= *table[i];  j++)
                          {
                     printf("%5d", table[i][j]);
                          }
printf("\n");
}
return 0; }
```

Lab Problem: Construct a ragged array

Sample input and output:

```
enter the number of rows: 3
enter size of row 1: 4
enter row 1 elements: 10 11 12 13
enter size of row 2: 5
enter row 2 elements: 20 21 22 23 24
enter size of row 3
enter row 3 elements: 30 31 32
displaying
10 11 12 13
20 21 22 23 24
30 31 32
```

END