COP 3502: Computer Science I
Spring 2004

– Day 4 –
Applications of Pointers in C

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2004

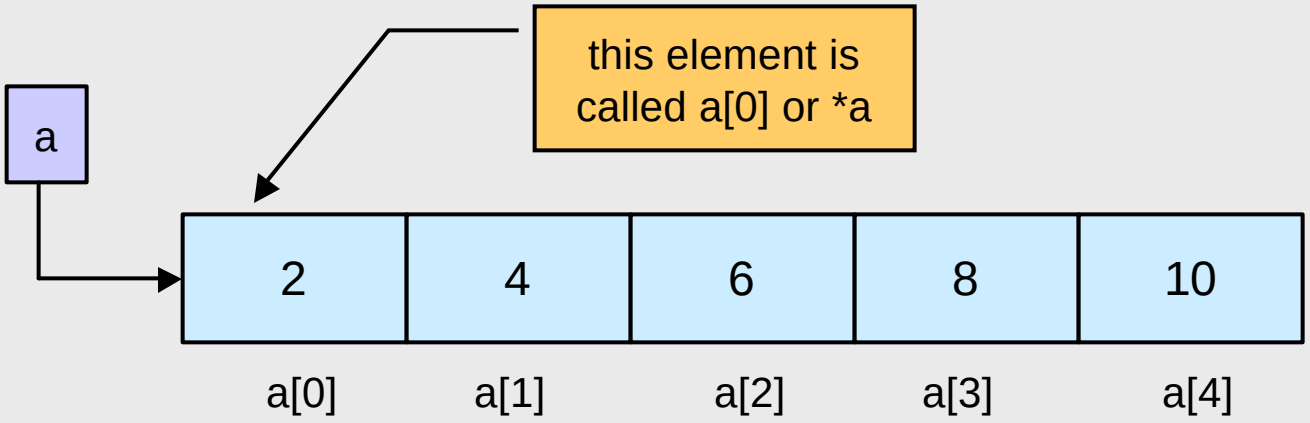
School of Electrical Engineering and Computer Science
University of Central Florida

Arrays and Pointers

- Arrays and pointers have a very close relationship. The name of an array is a pointer constant to the first element of the array.
- Because the array's name is a pointer constant, its value cannot be changed.
- Since the array name is a pointer constant to the first element, the address of the first element and the name of the array both represent the same location in memory.
- When an array name is de-referenced, it refers only to the first element, not the whole array.
  - Thus if a is the name of an array then a and &a[0] are the same.

Arrays and Pointers (cont.)

- Let's use the following array for a couple of examples.



This array was created with the following code:

```
int a[5] = {2, 4, 6, 8, 10};
```

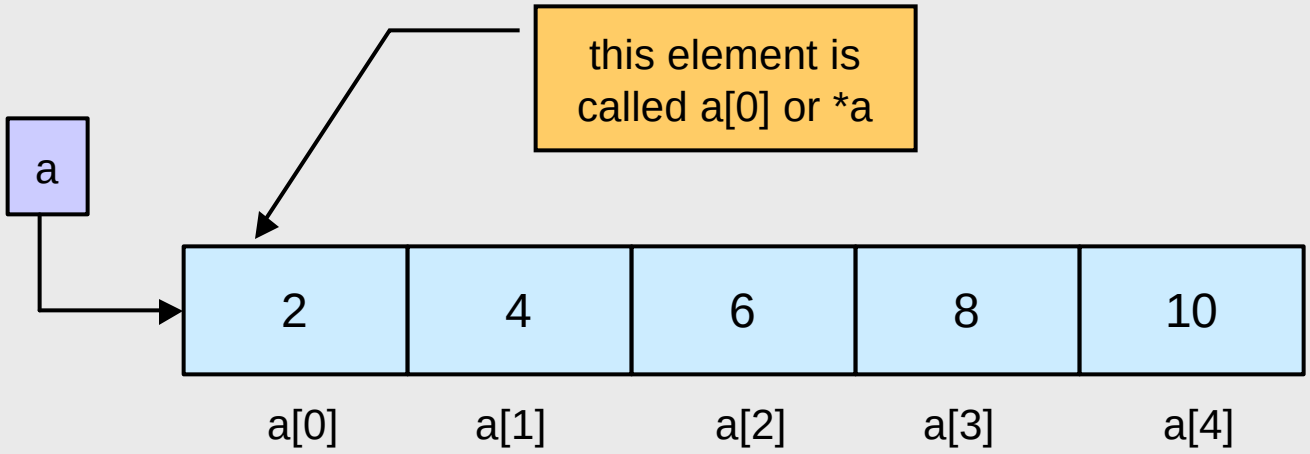

Example 1: De-reference of Array Name

Code

```
#include <stdio.h>
int main (void)
{   int a[5] = {2,4,6,8,10};
    printf("%d%d",*a, a[0]);
    return 0;
}
```

Output

2 2



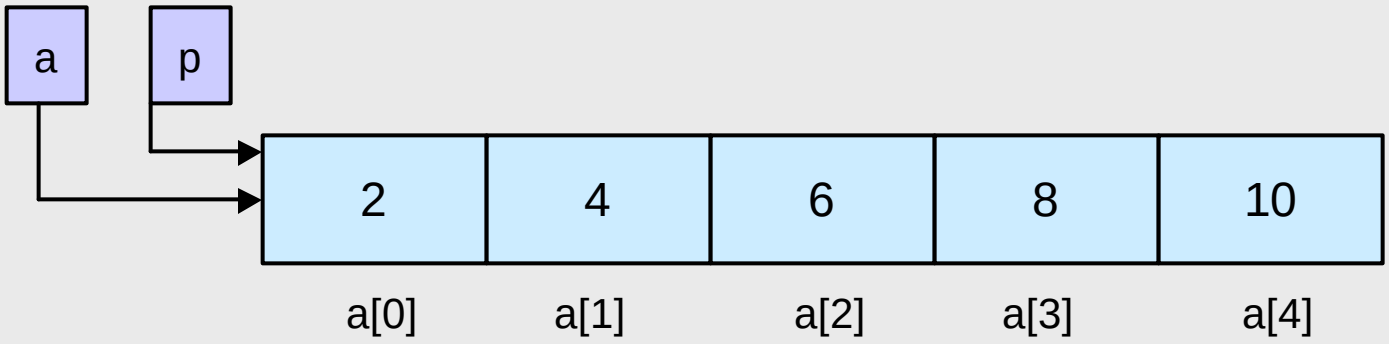
Example 2: Array Names as Pointers

Code

```
#include <stdio.h>
int main (void)
{   int a[5] = {2,4,6,8,10};
    int *p = a;
    int i = 0;
   printf("%d%d\n",a[i], *p);
   return 0;
}
```

Output

2 2



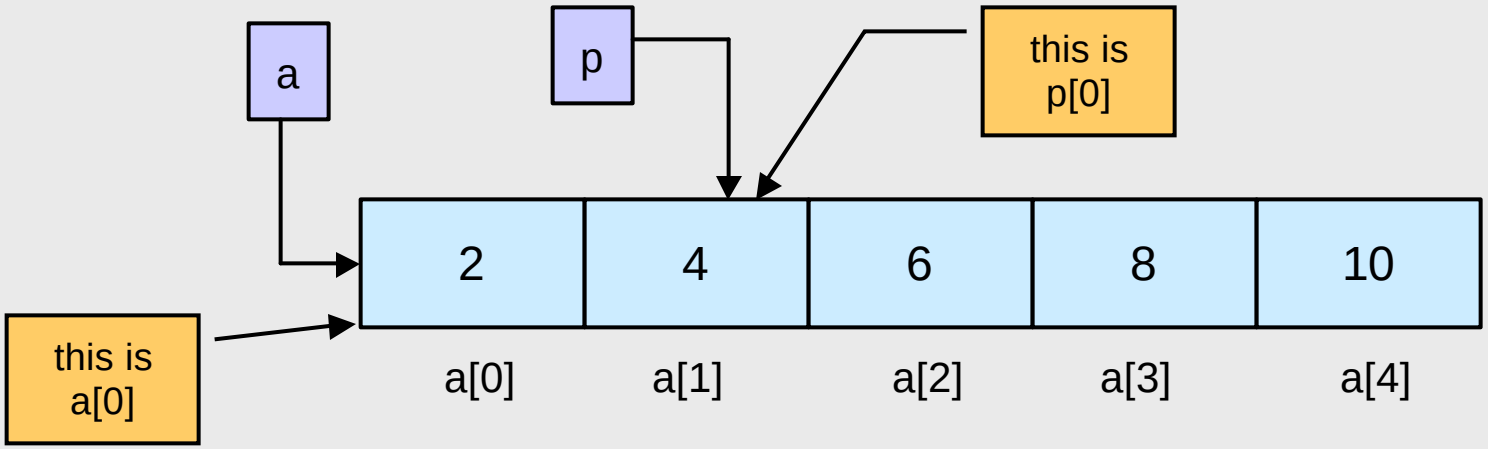
Example 2: Multiple Array Pointers

Code

```
#include <stdio.h>
int main (void)
{   int a[5] = {2,4,6,8,10};
    int p;
    p = &a[1];
   printf("%d%d\n", a[0], p[-1]);
   printf("%d%d\n", a[1], p[0]);
   return 0;
}
```

Output

2 2
4 4



Pointer Arithmetic and Pointers

- Besides indexing, another method for moving through an array is pointer arithmetic.
- Pointer arithmetic offers a restricted set of arithmetic operators for manipulating the addresses in pointers. It is especially powerful when you need to move through an array from element to element such as in a sequential search.
- As shown in the previous example, if we have an array named `a`, then `a` is a constant pointing to the first element and `a+1` is a constant pointing to the second element. Again, if we have a pointer `p`, pointing to the second element of an array, then `p-1` is a pointer to the previous (first) element and `p+1` would be a pointer to the next (third) element.

Pointer Arithmetic and Pointers (cont.)

- It does not matter how a and p are defined or initialized; as long as they are pointing to one of the elements of the array, we can add or subtract to get the address of the other elements in the array.

Given pointer p, p ± n is a pointer to the value n elements away.

- The meaning of adding or subtracting here is different from normal arithmetic. When you add an integer n to a pointer value, you will get a value that corresponds to another index location, n elements away. In other words, n is an offset from the original pointer.

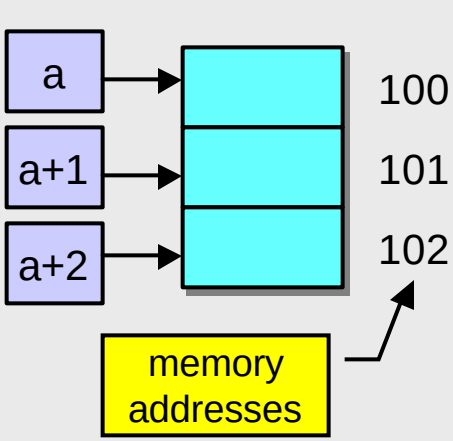
Pointer Arithmetic and Pointers (cont.)

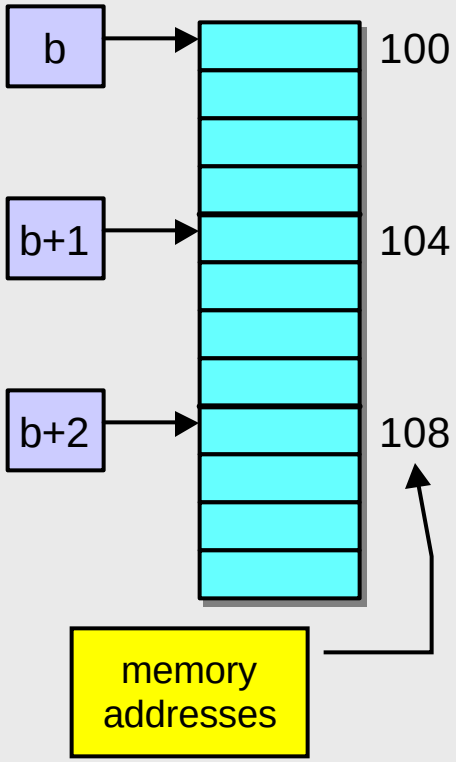
- To determine the new value, C must know the size of one element in the array. Recall that the size of the element is determined by the type of the pointer. This is one of the primary reasons that pointers of different types cannot be assigned to each other.
- If the offset is 1, then C can simply add or subtract one element size from the current pointer value. This may make the access more efficient that the corresponding index notation.
- If the offset is more than 1, then C must compute the offset by multiplying the offset by the element size and adding/subtracting it to/from the pointer value.

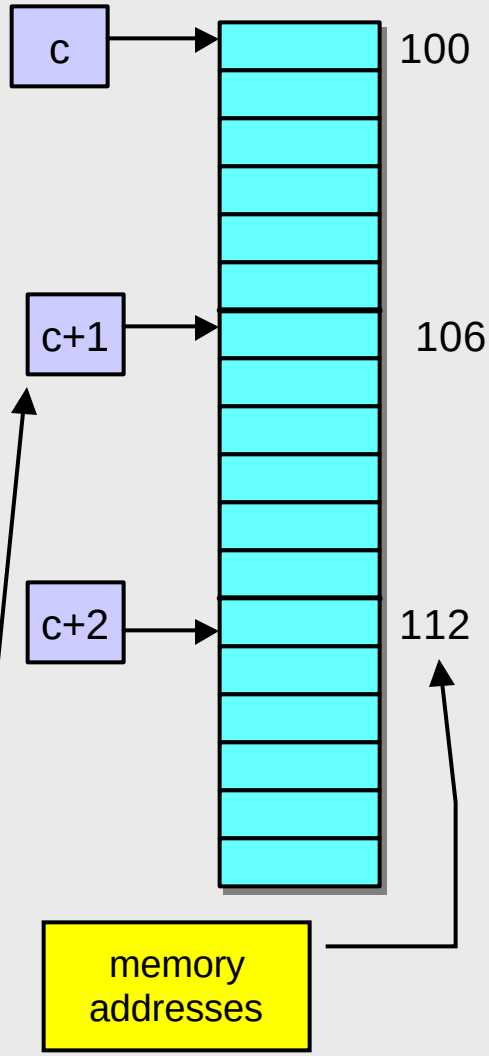
address = pointer + (offset * element size)

Example: Array Element Offsets

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



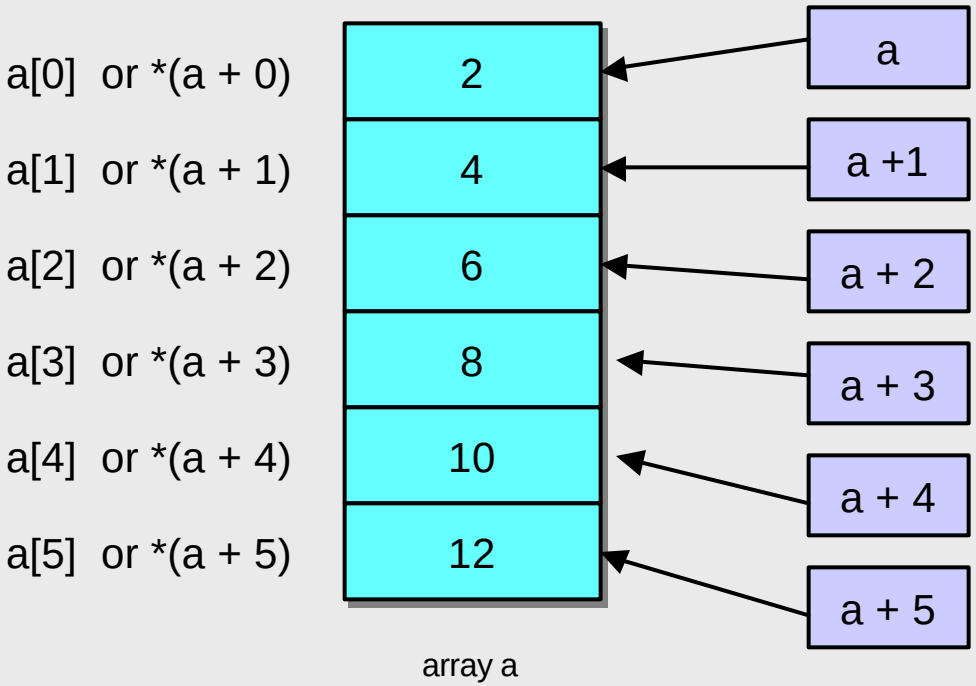




Assumption:
char = 1 byte
int = 4 bytes
float = 6 bytes

Adding 1 to c moves 6 bytes deeper into the memory. Similarly adding 2 moves 12 bytes deeper into memory.

Example: Dereferencing Array Pointers



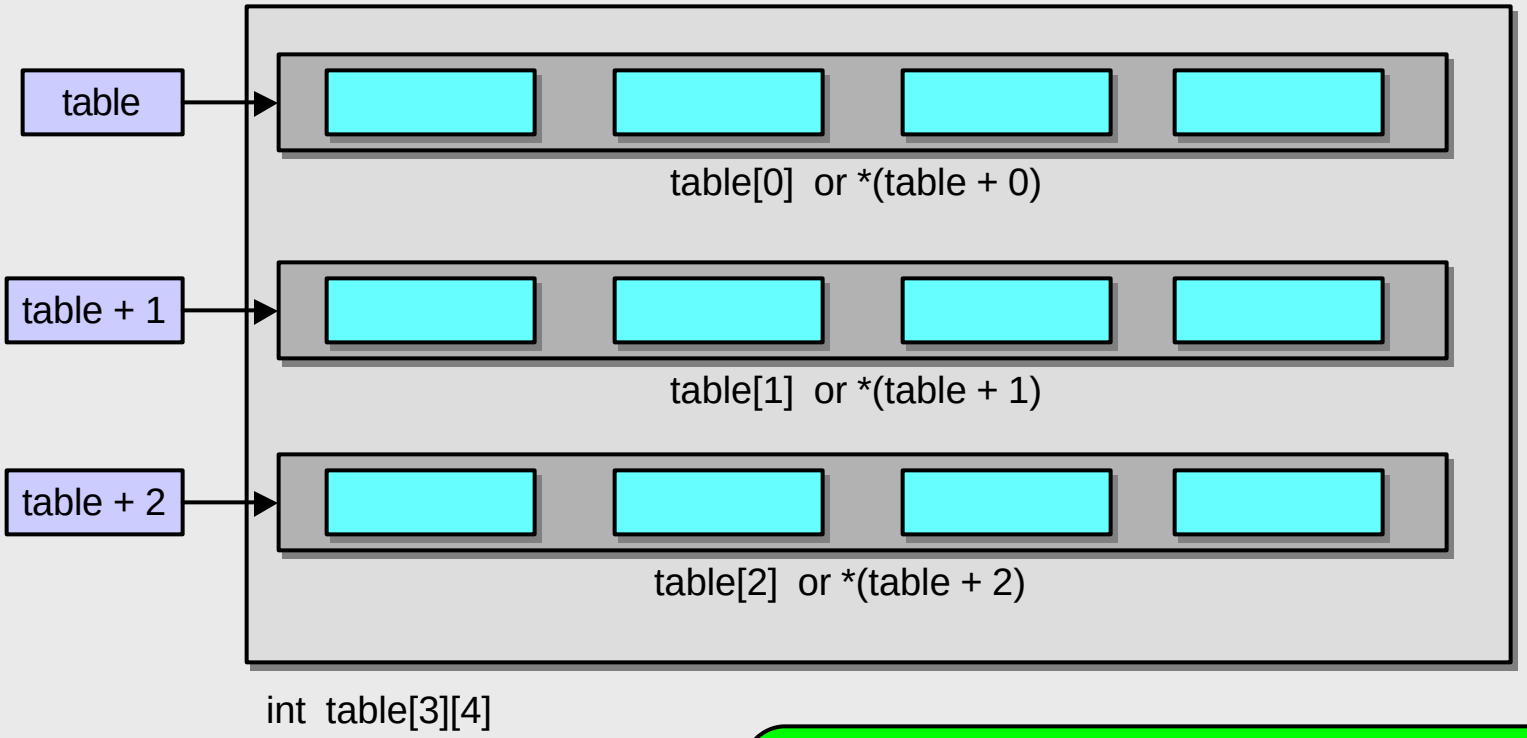
When a is the name of an array and n is an integer: *(a + n) ≡ a[n]

```
/* print an array in forward direction by adding 1 to a pointer and then print the */
/* same array in backwards direction by subtracting 1 from a pointer.              */
#include <stdio.h>
#define MAX_SIZE 10
int main (void)
{    int myarray = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
     int *pWalk;
     int *pEnd;
     /* forward direction */
     printf("Array forward: ");
     for (pWalk = myarray, pEnd = myarray + MAX_SIZE; pWalk < pEnd; pWalk++)
          printf("%3d", *pWalk);
     printf("\n");
     /* backward direction */
     printf("Array backward: ");
     for(pWalk = pEnd-1; pWalk >= myarray; pWalk--)
          printf("%3d", *pWalk);
     printf("\n");
     return 0;
} /*end main */
```


Pointers and Two-Dimensional Arrays

- Just as was the case with one-dimensional arrays, the name of the array is a pointer constant to the first element of the array. In the case of a 2d array, the first element is another array.
- Assuming that we have a 2d array of integers, when we dereference the array name, we don't get one integer, we get an array of integers.
  - In other words, the dereference of the array name of a 2d array is a pointer to a one dimensional array.

Pointers and Two-Dimensional Arrays
(cont.)



table[0] is identical to *(table + 0).

table[0][0] is identical to *(*(table)).

Pointers and Two-Dimensional Arrays (cont.)

- When dealing with multi-dimensional arrays, there is no simple pointer notation.
- To refer to a row, dereference the array point, which gives a pointer to a row.
  - example: table[0] ≡ *(table + 0)
- Given a pointer to a row, dereference the row pointer, which gives a pointer to an individual element.
  - example: table[0][0] ≡ *(*(table))
- The code examples on the next page illustrate how to print the elements of a 2-d array using both index and pointer notation.

```
/* print a 2-d array using index notation */
/* using example array from page 14     */
…
for (i = 0; i < 3; i++)
{   for (j = 0; j < 4; j++)
        printf("%6d", table[i][j]);
    printf("\n");
}  /* end for loop on i */
…
```

```
/* print a 2-d array using pointer notation */
/* using example array from page 14       */
…
for (i = 0; i < 3; i++)
{   for (j = 0; j < 4; j++)
        printf("%6d", *(*(table + i) +j));
    printf("\n");
}  /* end for loop on i */
…
```

With multi-dimensional arrays, pointer arithmetic has no efficiency advantage over indexing. Because pointer notation for multi-dimensional arrays is so complex and there is no efficiency advantage, most programmers find it easier to use the index notation.

Passing an Array to a Function

- Since we know that the name of an array is actually a pointer to the first element, we can send the array name to a function for processing.
- When passing the array, do not use the address operator. Since the array name is a pointer constant, it is already an address to the first element of the array.
- An example of a typical call is: doIt (myarray);

Passing an Array to a Function (cont.)

The called program can declare the array in one of two ways.

1. It can use the traditional array notation. This format has the advantage of telling the user very clearly that we are dealing with an array rather than a single pointer.
   - example: `int doIt (int myarray[ ])`
2. It can also declare the array in the function header as a simple pointer. The disadvantage of this format is that, while it is technically correct, it actually masks the data structure. However, for one-dimensional arrays it is very commonly used, although a good descriptive name for the parameter should be used to indicate it is an array that is being referenced.
   - example: `int doIt (int *anArray)`

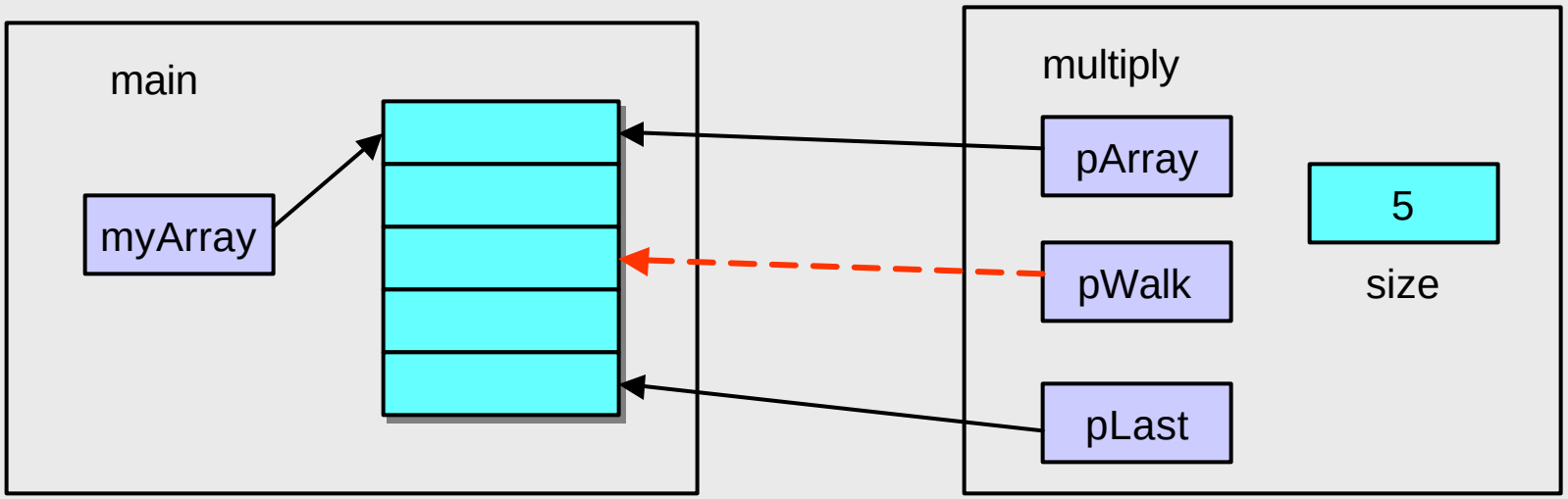
Passing an Array to a Function (cont.)

- If a multi-dimensional array is being passed as a parameter to a function, you must use the array notation in the header declaration and definition.
- The compiler needs to know the size of the dimensions after the first to calculate the offset for pointer arithmetic.
- Thus, for a 3-d array, you would use the following declaration in the function's header:

```
float doIt (int bigArray [ ] [12] [7])
```

Example: Passing an Array to a Function

- Let's look at an example of passing an array to a function. In the code on the next page, the function multiplies each element of a passed array by 2.
- The variables needed are shown below:



```
/* read integers from the keyboard into an array and then print the array with the */
/* values doubled from the original input values.                                   */
#include <stdio.h>
#define SIZE 5
/* PROTOTYPES */
void multiply (int *pArray, int size);
int main (void)
{     int myArray[SIZE];
      int *pWalk;
      int *pLast;
      pLast = myArray + SIZE – 1;
      for (pWalk = myArray; pWalk <= pLast; pWalk++)
      {    printf(“Please enter an integer number: ” );
           scanf(“%d”, pWalk);
      }
      multiply (myArray, SIZE);
      printf(”Doubled value of array elements are: \n “);
      for(pWalk = myArray;  pWalk <= pLast;  pWalk++)
                printf(“%3d”, *pWalk);
      return 0;
} /*end main */
```

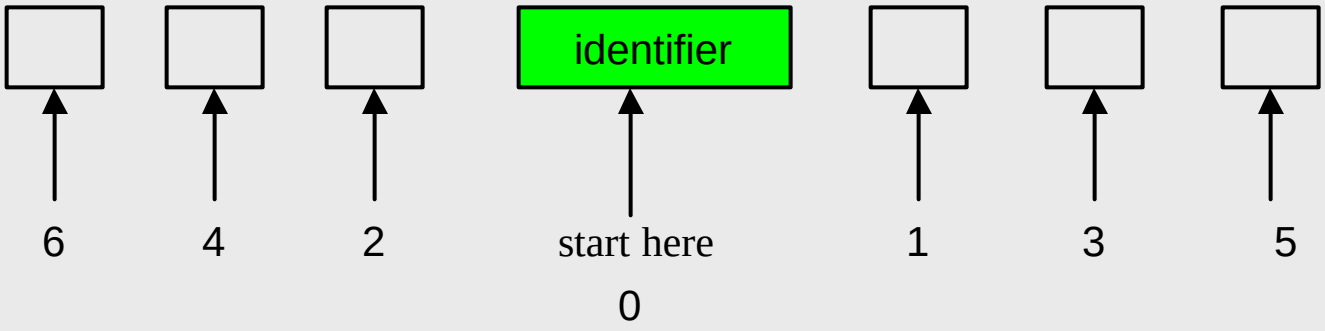

```
/* MULTIPLY FUNCTION                                                    */
/* Assumes array is filled with integer values.   SIZE is number of elements    */
/* Doubles the values in the array.                                     */
void multiply (int *pArray, int size)
{     int *pWalk;
      int *pLast;
      pLast = pArray + size – 1;
      for (pWalk = pArray; pWalk <= pLast; pWalk++)
      {
            *pWalk = *pWalk * 2;
      }
     return;
} /*end multiply */
```


Deciphering Complex Declarations

- As data structure definitions become more complex, their declarations become increasingly complicated. Sometimes declarations are difficult to interpret, even for someone well experienced in the C language.
- To help you read and understand complicated declarations, remember the right-left rule.
- Using the right-left rule to interpret a declaration, you start with the identifier in the center of a declaration and read the declaration by alternatively going right and then left until you have read all the entities.

Deciphering Complex Declarations
(cont.)

- The figure below graphically illustrated the right-left rule.



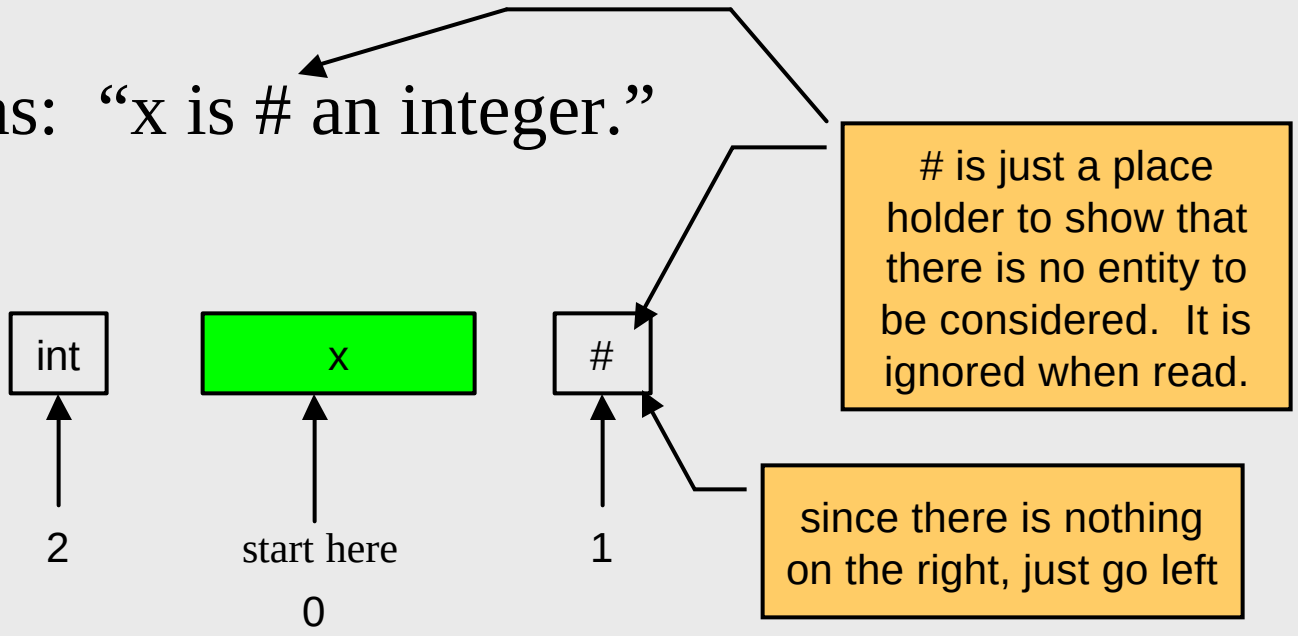
Examples: Deciphering Complex Declarations

- Let's look at some simple examples to begin with before trying more complex examples.

1. Declaration: int x

   Read as: "x is # an integer."

   Why:

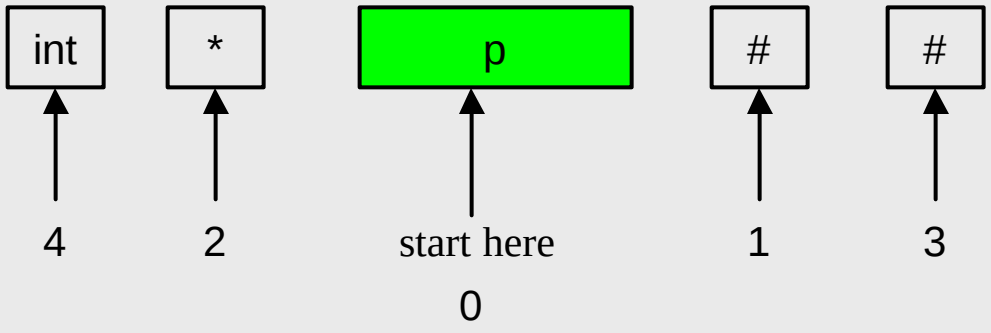


Examples: Deciphering Complex Declarations (cont.)

2. Declaration: int *p

Read as: "p is # a pointer # to integer."

Why:

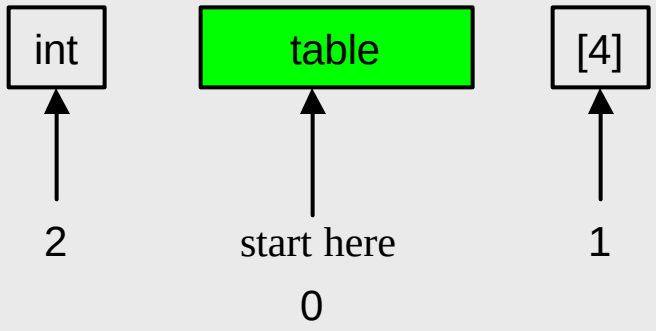


Examples: Deciphering Complex Declarations (cont.)

3. Declaration: int table [4]

   Read as: "table is an array of 4 integers."

   Why:

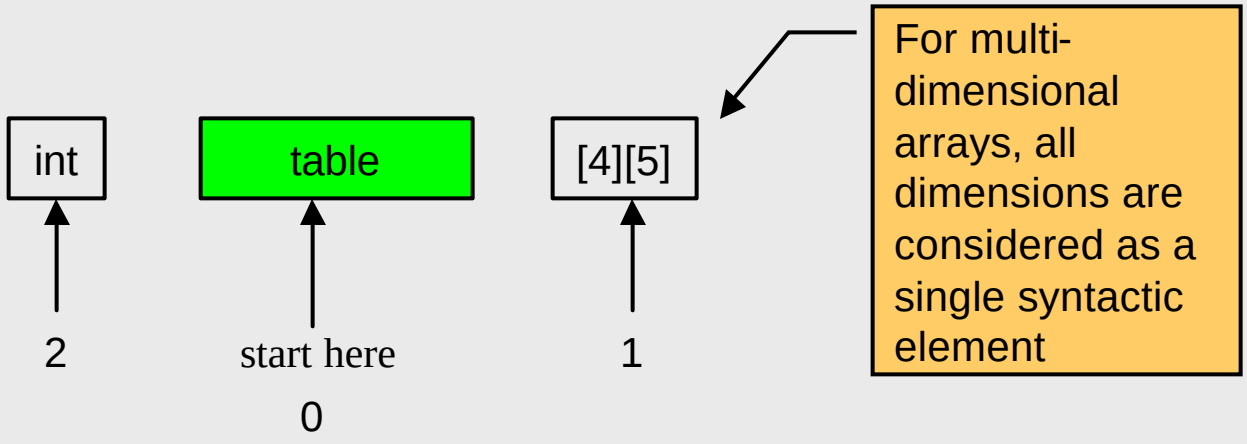


Examples: Deciphering Complex Declarations (cont.)

4. Declaration: int table [4][5]

   Read as: "table is a [4][5] array of integers."

   Why:

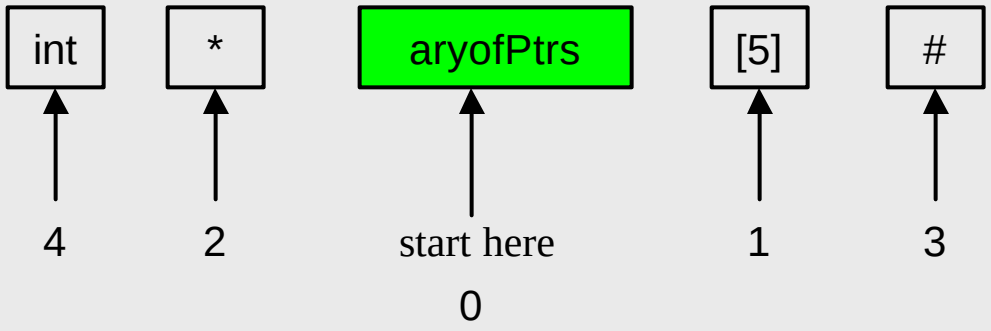


Examples: Deciphering Complex Declarations (cont.)

5. Declaration: int *aryofPtrs[5]

   Read as: "aryofPtrs is an array of 5 pointers to # integer."

   Why:

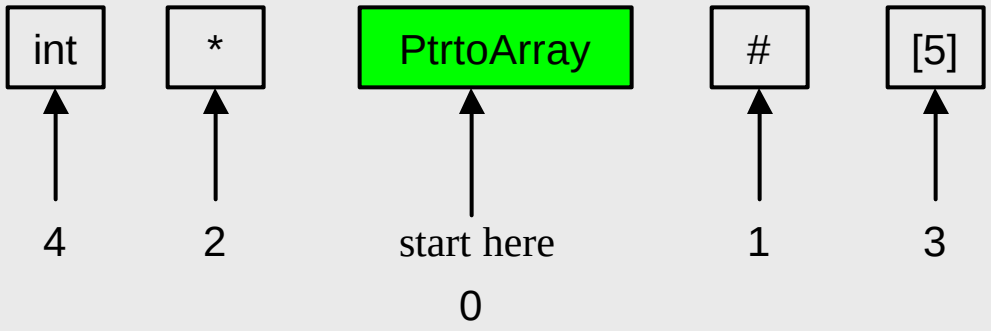


Examples: Deciphering Complex Declarations (cont.)

6. Declaration: int (*PtrtoArray)[5]

   Read as: "PtrtoArray is a # pointer to an array of 5 integers."

   Why:

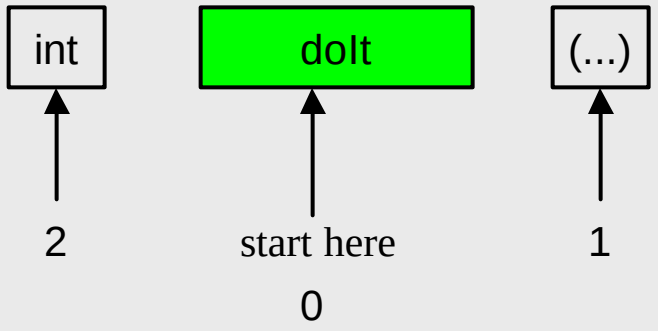


Examples: Deciphering Complex Declarations (cont.)

7. Declaration: int doIt (...)

   Read as: "doIt is a function returning an integer."

   Why:

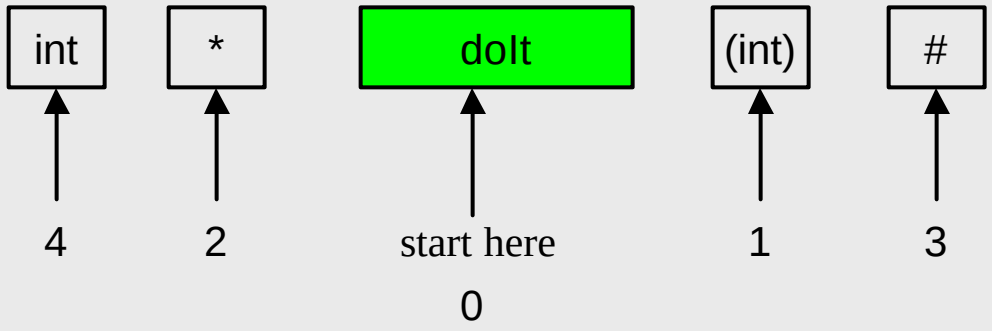


Examples: Deciphering Complex Declarations (cont.)

8. Declaration: int * doIt (int)

   Read as: "doIt is a function returning a pointer to # an integer."

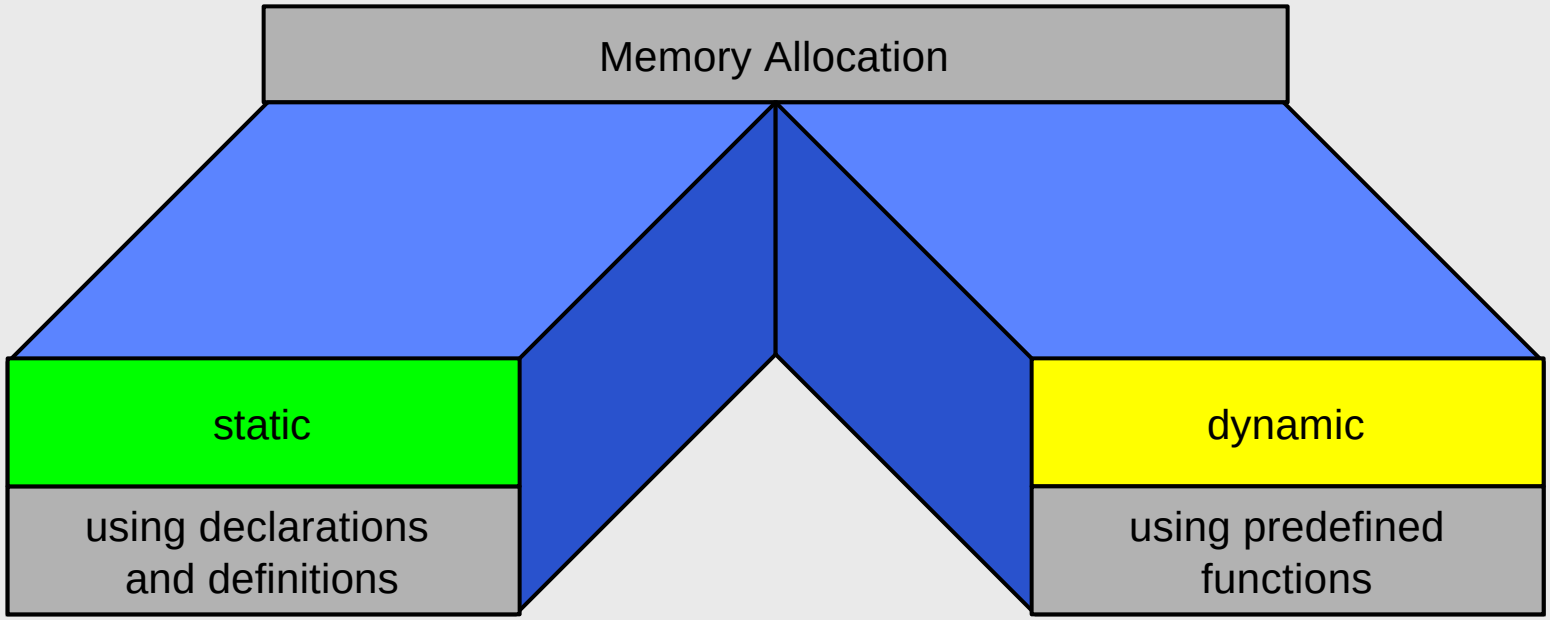
   Why:



Memory Allocation Functions

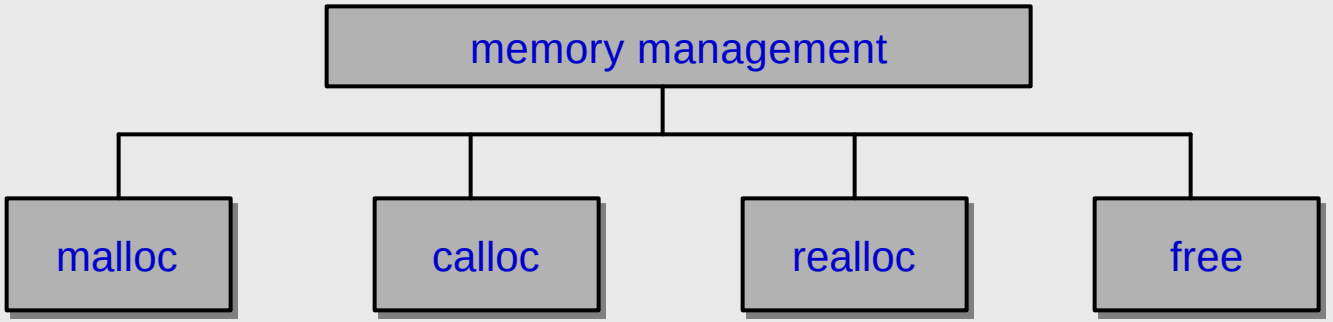
- Static memory allocation requires that the declaration and definition of memory be fully specified in the source program. The number of bytes reserved cannot be changed during execution.
- Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is in execution. This effectively postpones data definition until run time.

Memory Allocation (cont.)



Memory Allocation Functions (cont.)

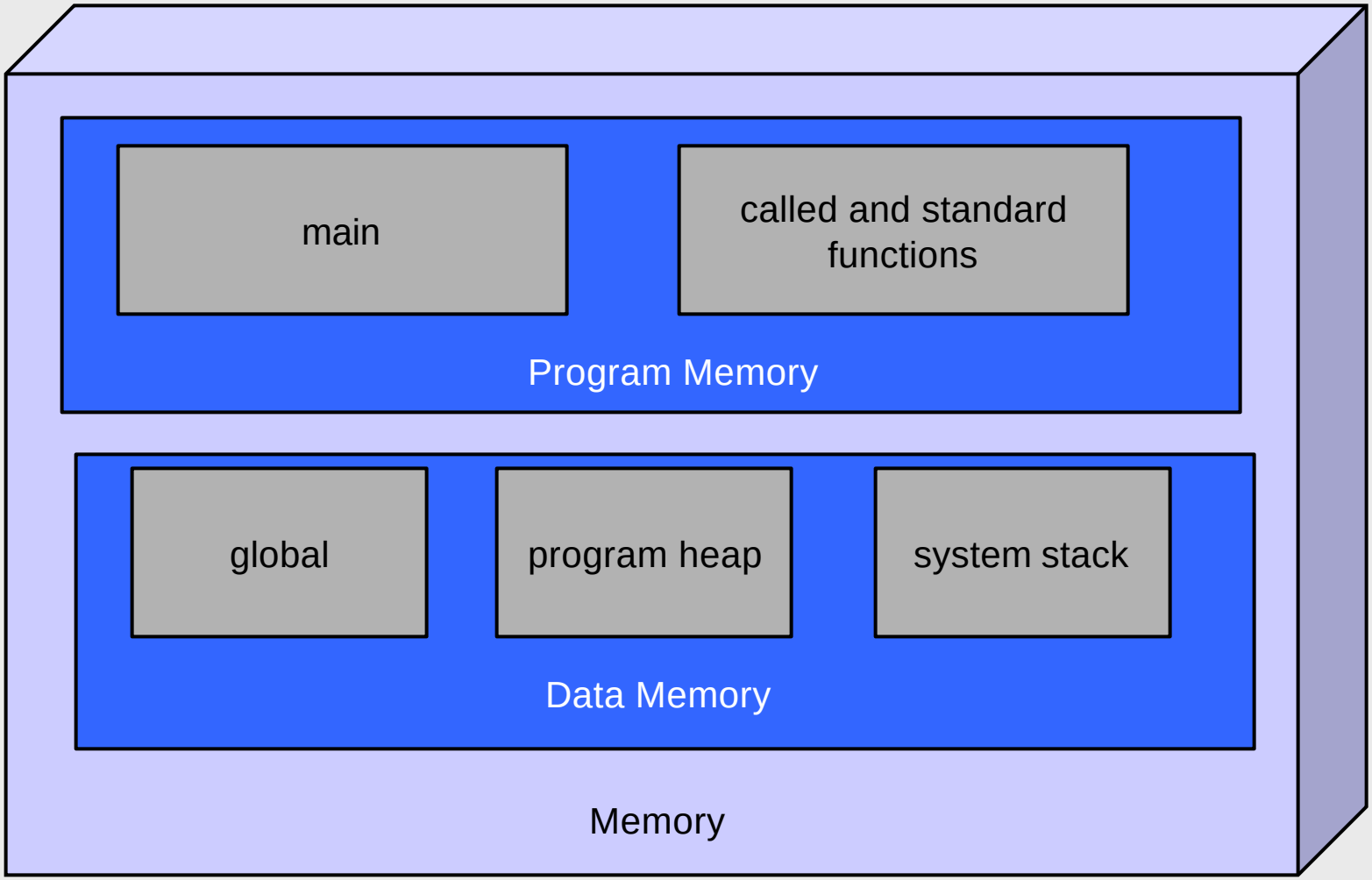
- Four memory management functions are used with dynamic memory in the C language.
  - malloc, calloc, and realloc are used for memory allocation.
  - free is used to return allocated memory to the system when it is no longer needed.
- All the memory management functions are found in the standard library header file <stdlib.h>.



Memory Allocation Functions (cont.)

- Conceptually, memory is divided into:
  - program memory which is used for `main` and all called functions, and
  - data memory which is used for global data, constants, local definitions and dynamic memory.
- Obviously, `main` must be in memory at all times. Each called function must only be in memory while it or any of its called functions are active. Since multiple copies of a function may be active at one time (recursion) the multiple copies of the variables are maintained on the stack. The heap memory is unused memory allocated to the program and available to be assigned during execution.
- The next page illustrates the conceptual view of memory.

Conceptual View of Memory

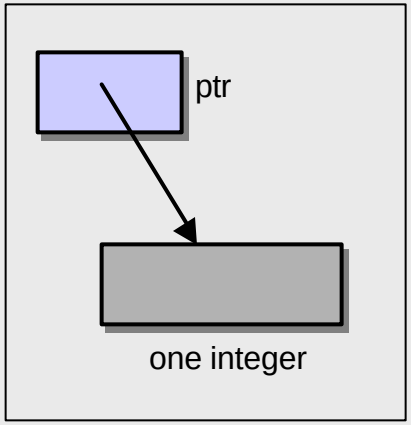


Memory Allocation: malloc

- The malloc function allocates a block of memory that contains the number of bytes specified in its parameter. It returns a void pointer to the first byte of the allocated memory.

Typical malloc call:

```
if (!(ptr = (int *) malloc(sizeof(int))))
    /*no memory available */
    exit(100);
/*memory available */
...
```



Memory Allocation: calloc

- The calloc function is primarily used to allocate memory for arrays. It differs from malloc in three ways:

1. It allocates a contiguous block of memory large enough to contain an array of elements of a specified size. It requires two parameters, the first for the number of elements to be allocated and the second for the size of each element.

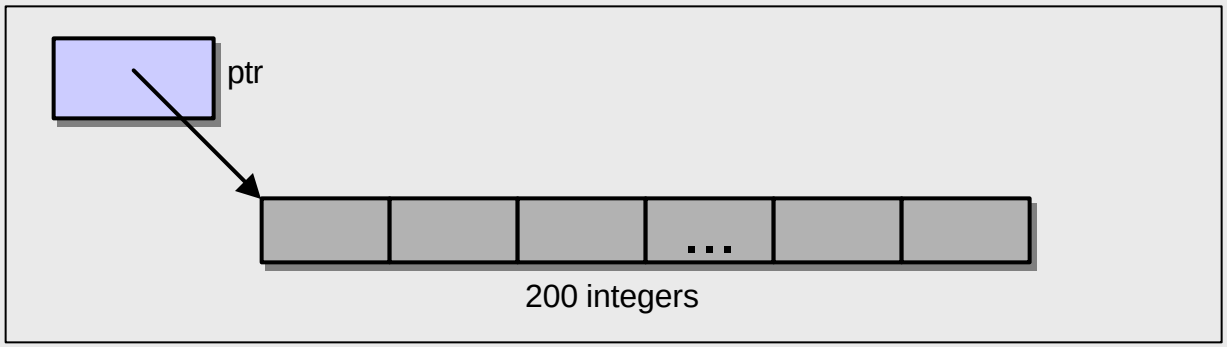
2. It returns a pointer to the first element of the allocated array. Since it is pointing to an array, the size associated with its pointer is the size of one element, *not the entire array.*

3. It clears memory by setting each location in the array to 0, although 0 is not guaranteed to be the null value for types other than integer.

Memory Allocation: calloc (cont.)

Typical calloc call:

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

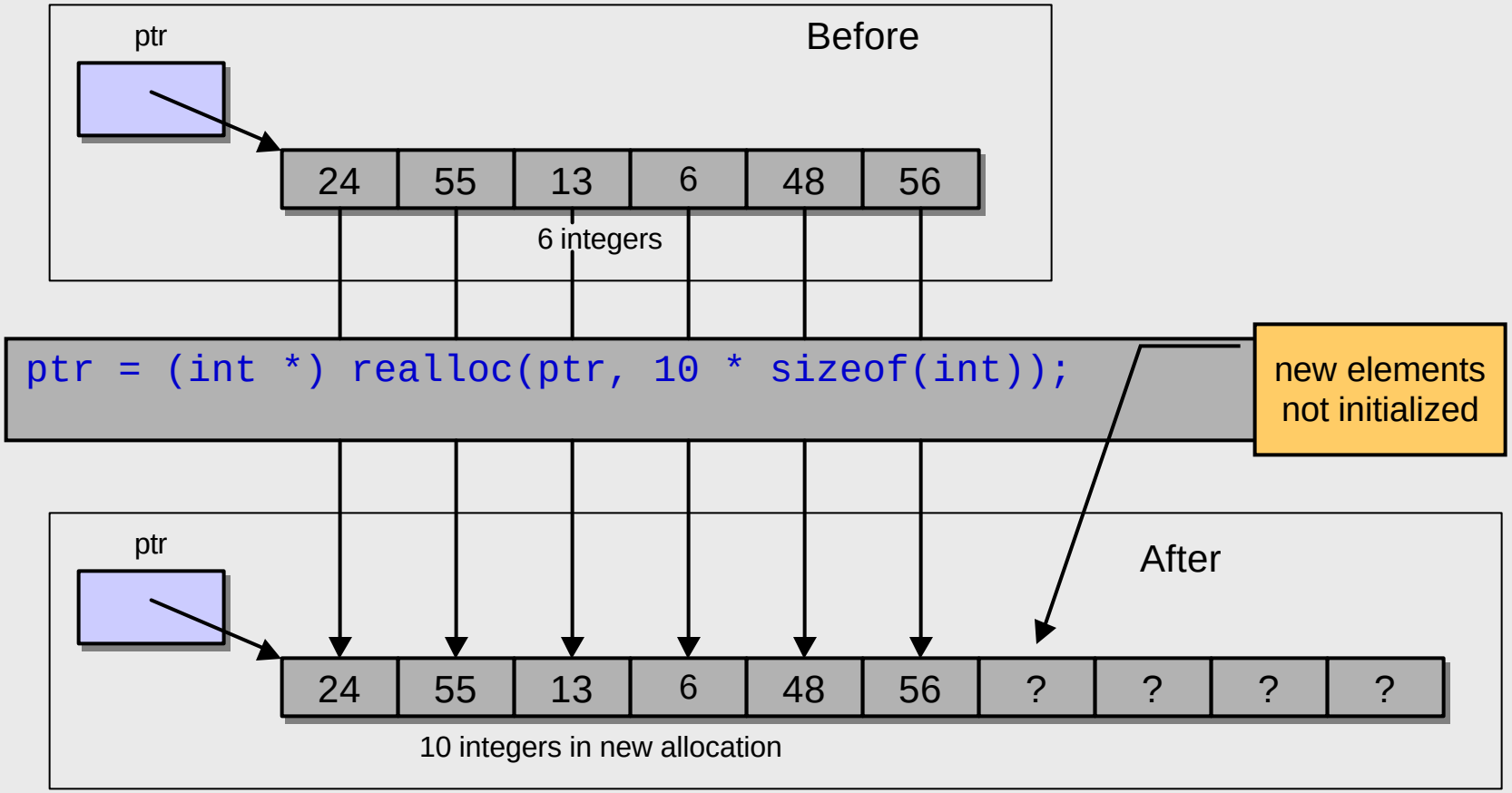


Memory Allocation: realloc

- The realloc function can be highly inefficient and therefore should be used advisedly. When 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 because of 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 any other pointers to the data are correctly changes.

Memory Allocation: realloc (cont.)

Typical realloc call:



Memory Allocation: free

- When memory locations allocated by malloc, calloc, or realloc are no longer needed, they should be freed using the free function.
- Note that it is not the pointers that are being released but rather what they point to.
- To release an array of memory allocated by calloc, you need only to release the pointer once. It is an error to attempt to release each element individually.
- Releasing memory does not change the value in a pointer. It still contains the address in the heap. It is a logic error to use the pointer after memory has been released. It is good programming practice to clear the pointer by setting it to NULL immediately after the memory is released.