# Pointer and Arrays

Instructor: HOU, Fen

2025

# Motivation to Introduce Arrays

- Programs often use homogeneous data such as the grades.
- How to organize the grades of 100 students?
  - 100 different identifiers for each course. Such as, int grade0, grade1, grade2
  - It is cumbersome to manipulate these data by means of unique identifiers. Then we use a derived type named as array
- Array
  - An efficient way to organize data with the same type, such as int grade[5];
  - A single name for a collection of data values.

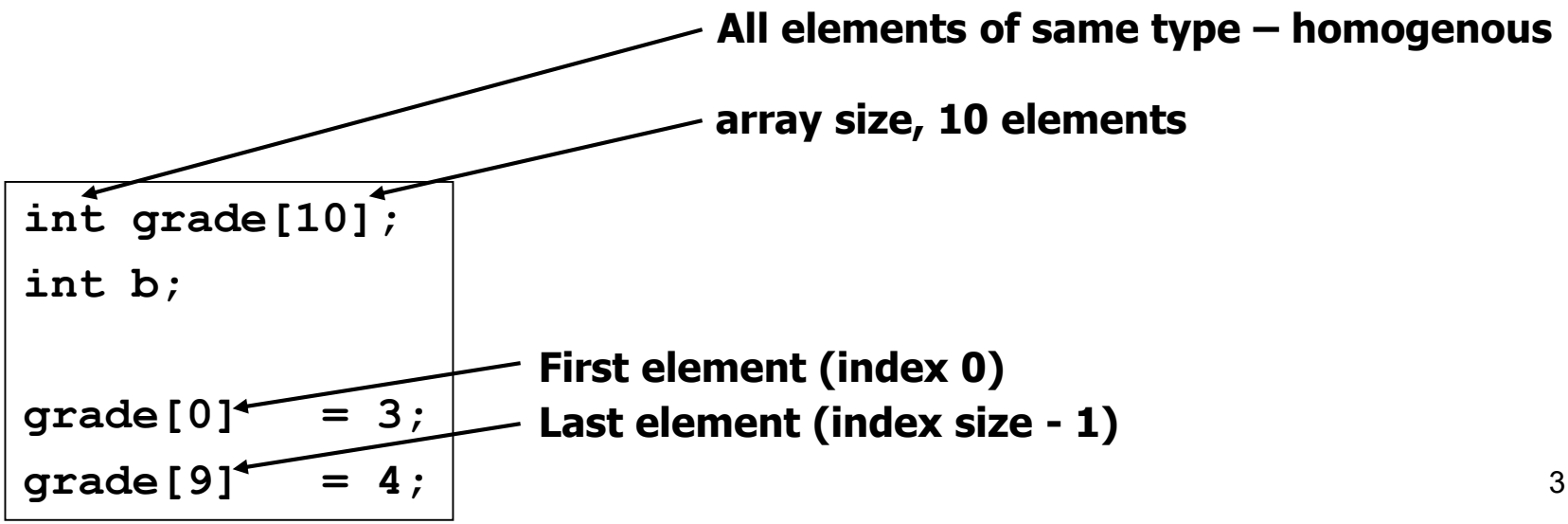
# Array Declaration

- Syntax for declaring a one-dimensional array

  ***Date_Type Array_Name[Length];***

  - Length is an int value, which indicates the size of the array, that is the number of elements in the array.
  - Individual elements of the array are accessed using a subscript, also called as index.
  - The brackets[] are used to contain the subscripts of an array. That is, the index of an array.

- The value of a subscript/index must be a no negative integer within the range from 0 to (size-1)

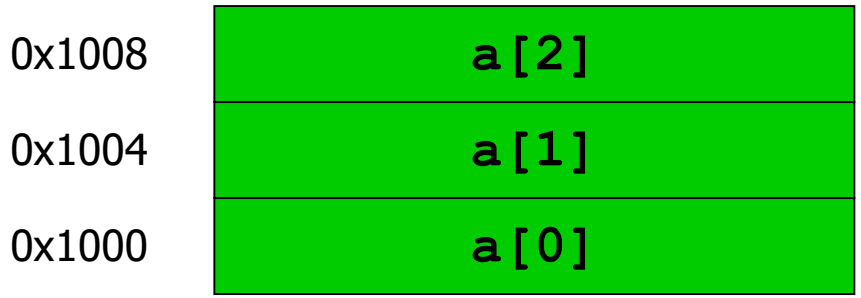


# Arrays

- Homogeneous: Each element has the same size (the size depends on the data type and the computer). For example, when the data type is int, the size of each element could be the size of 4 bytes.
- To store the element of an array, the complier assigns an appropriate amount of memory, starting from a base address.
- The indexing of array elements always starts at 0;
- Array size:

```
int a[3];
sizeof(a[0]);      /*the result is 4 */
sizeof(a);   /* the result is 12  */
```

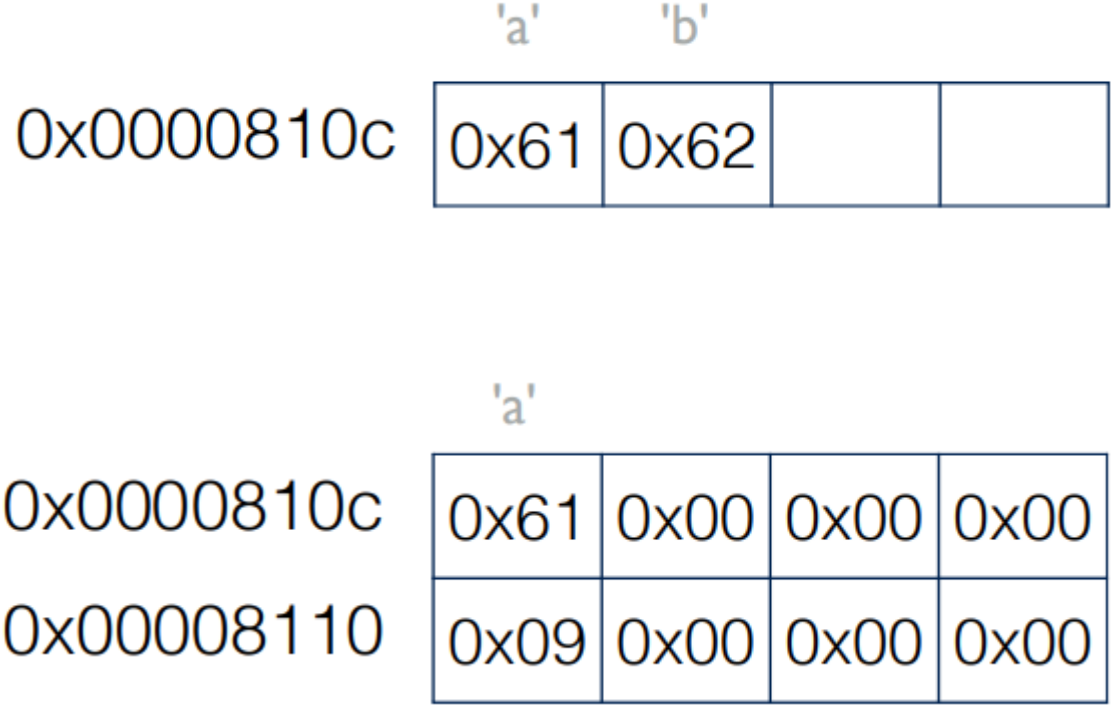
**`int a[3];`**



# Arrays

- An array allocates multiple instances of the same type contiguously in memory
- For example

```
char ab[2];
ab[0]='a';
ab[1]='b';
int x[2];
x[0]='a';
x[1]=9
```



# Array Initialization

- Arrays can be initialized within a declaration using a sequence of values written as a brace-enclosed, comma-separated list.

  float grade[3]= {73, 99, 85};

  int x[4] ={1,3,9,2};

- If an array is declared without a size and is initialized to a series of values. It is implicitly given the size of the number of initializers. For example, the following declarations are equivalent.

  int a[]={2,4,7}; and int a[3] ={2,4,7};

# Array Initialization

- Arrays can be filled after the declaration

```
/*Fill and Print an array. */
#include <stdio.h>

#define   N   5

int main(void)
{
   int   a[N];       /* allocate space for a[0] to a[4] */
   int   i, sum = 0;

   for (i = 0; i < N; ++i)           /* fill the array */
      a[i] = 7 + i * i;
   for (i = 0; i < N; ++i)           /* print the array */
      printf("a[%d] = %d    ", i, a[i]);
   for (i = 0; i < N; ++i)           /* sum the elements */
      sum += a[i];
   printf("\nsum = %d\n", sum);     /* print the sum */
   return 0;
}
```

- Output:

```
a[0] = 7    a[1] = 8    a[2] = 11   a[3] = 16    a[4] = 23
sum = 65
```

# Exercise: Calculate Average

Code:

```
#include <stdio.h>
int main() {
  int marks[10], i, n, sum = 0;
  double average;
  printf("Enter number of elements: ");
  scanf("%d", &n);
  for(i=0; i < n; ++i)
 {
    printf("Enter number%d: ",i+1);
    scanf("%d", &marks[i]);
   sum += marks[i];
  }
  average = (double) sum / n;
  printf("Average = %.2lf", average);

  return 0;
}
```

Output:

```
Enter number of elements: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39.60
```

# The Relationship between Arrays and Pointers

- An array name by itself is a pointer constant to the first element of this array.
- The address of the first element of an array and the array name both represent the same location in memory
- int a[5], *p, then the following two statements are equivalent

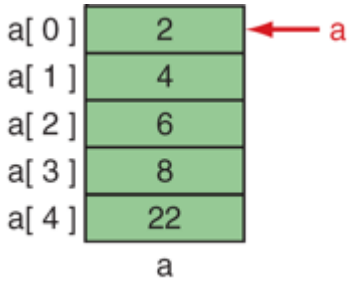
  p=a and p=&a[0]


- That is, a is equivalent to &a[0]
- The first element is *a or a[0]

# The Relationship between Arrays and Pointers

- Since the array name is a pointer constant, and NOT a variable. We cannot change the value of a.
- Illegal expressions: a=p; ++a; a+=2;
- Example

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




OUTPUT:

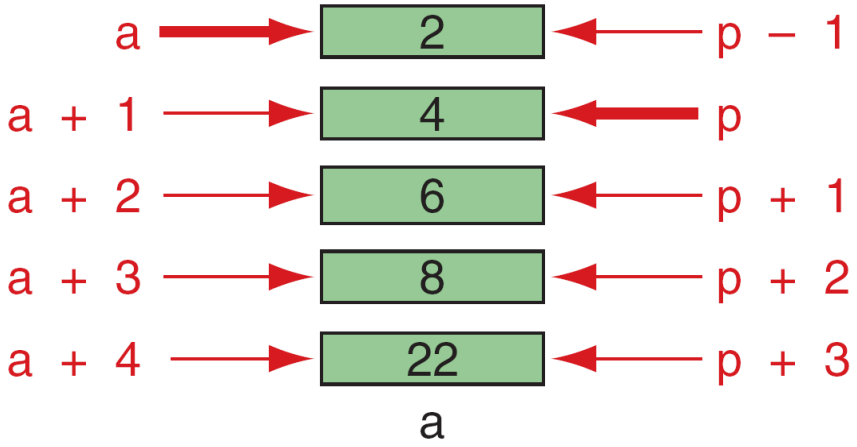
2 2

# Arrays and Pointers

- Example, assign an array name to a pointer

```
int a[2] = {5, 7};   //int 32 bits, 4 bytes
int * p= a;     // p = &(a[0]);
```

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

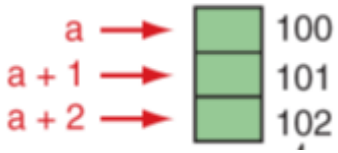


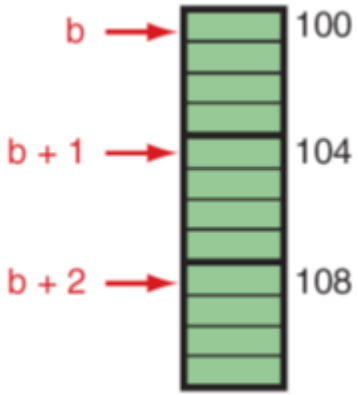

- The expression *(a+i) is equivalent to a[i];

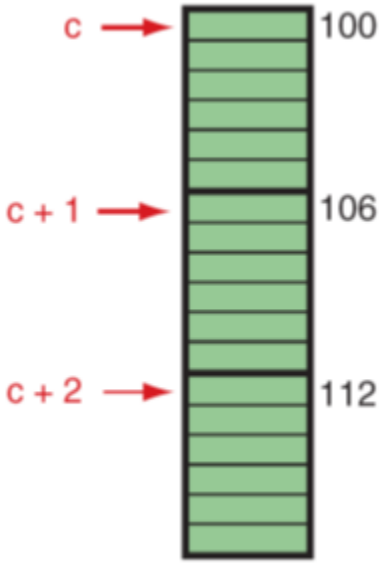
# Arrays and Pointers

- Incrementing pointers advances address by size of type .
- Example;

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



# Arrays and pointers

Given int n, arr[4], *p, which of the following is illegal? Why?

A. p = arr; p = &arr[0];

B. arr = p;

C. *p = 3; p[0] = 3;

D. n = *(arr + 1);

E. n = arr[1];

- Answer:

  B is illegal.

# Example: Sum the array

- Code

```
#define   N   100

int   a[N], *p;

sum = 0;
for (p = a; p < &a[N]; ++p)
   sum += *p;
```

- Equivalent code

```
#define   N   100

int   a[N], *p;

sum = 0;
for (i = 0; i < N; ++i)
   sum += *(a + i);
```

# Example: Sum the array

- Code

```
#define   N   100

int   a[N], *p;

sum = 0;
for (p = a; p < &a[N]; ++p)
   sum += *p;
```

- Equivalent code

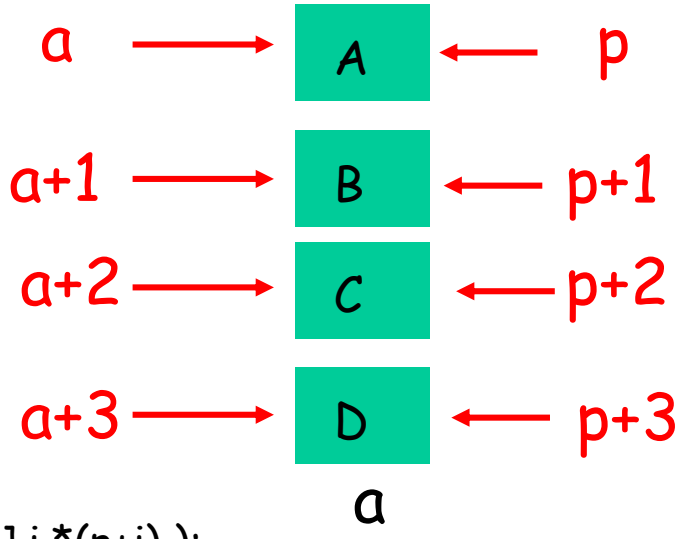
```
#define   N   100

int   a[N], *p;

p = a;
sum = 0;
for (i = 0; i < N; ++i)
   sum += p[i];
```

- In many ways, arrays and pointers can be treated alike. The following expressions are equivalent
  a[i] and *(a+i), and p[i]
- there is one essential difference: The array a is a constant pointer, Not a variable.
- Illegal:
  a=p ;
  ++a ;
  a+=2 ;

# Arrays and Pointers

- Assign an array name to a pointer p
- Given pointer p, p ± n is a pointer to the value n elements away.
- The expression *(a+i) is equivalent to a[i];
- Example

```
#include <stdio.h>
Main ()
{
int i, *p;
char a[4]={'A','B','C','D'};
p=a;
for (i=0;i<4;i++)
printf("a[%d] value:%c; *(p+%d) value:%c\n"; i, a[i],i,*(p+i) );
}
```




- Output

# Exercise

- Please write a program to print an array forward by adding 1 to a pointer. Then print it backward by subtracting 1.

```
#include <stdio.h>

#define MAX_SIZE 10

int main (void)
{
// Local Declarations
   int  ary[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int* pWalk;
   int* pEnd;

printf("Array forward : ");
for (pWalk = ary, pEnd = ary + MAX_SIZE;
     pWalk <  pEnd;
     pWalk++)
  printf ("%3d", *pWalk);
printf ("\n");

// Print array backward
printf ("Array backward: ");
for (pWalk = pEnd - 1; pWalk >= ary; pWalk--)
  printf ("%3d", *pWalk);
printf ("\n");

return 0;
}  // main
```

- Output:

```
Array forward :   1  2  3  4  5  6  7  8  9 10
Array backward:  10  9  8  7  6  5  4  3  2  1
```

# Exercise

- What is the output of the following code?

```
#include <stdio.h>
#define  N  5
int main()
{
  int i, * ptr, sum = 0;
  int nums[N] = {1, 2, 3, 4, 5};
  for(i = 0; i < N; ++i)
     sum += *(nums+i);
  printf("Sum = %d ", sum);}
```

- Output: 15

# Exercise

- What is the output of the following code?

```
#include <stdio.h>
#define N 5
int main() {
  int i, * ptr, sum = 0;
  int nums[N] = {1, 2, 3, 4, 5};
  ptr = nums;
  for(i = 0; i < N; ++i)
     sum += ptr[i];
  printf("Sum = %d ", sum); // Sum = 15
}
```

- Output: 15

# Exercise

- Read the following code.

```
#include <stdio.h>

int nums[] = {0, 5, 87, 32, 4, 5};
int *ptr;
int main(void)
{
int i;
ptr = &nums[0]; /* pointer to the first element of the array */
printf("Print array elements using the array notation\n");
printf("and  by dereferencing the array pointers:\n");
for (i = 0; i < 6; i++)
{
printf("\n nums[%d] = %d ", i , nums[i]);
printf("\n ptr + %d = %d\n",i, *(ptr + i));
}
return 0;
}
```

```
Print array elements using the array notation
and  by dereferencing the array pointers:

nums[0] = 0
ptr + 0 = 0

nums[1] = 5
ptr + 1 = 5

nums[2] = 87
ptr + 2 = 87

nums[3] = 32
ptr + 3 = 32

nums[4] = 4
ptr + 4 = 4

nums[5] = 5
ptr + 5 = 5
```

# Two Dimensional Array

- Syntax for variable declaration

  data_type ArrayName[rowsize][columnsize]
  - Data_type: a valid data type like int, char, or float
  - ArrayName: a valid identifier
  - Rowsize and columnsize: maximum no. of elements that can be stored in the row and the column
- Initialization

  int arr[2][2];

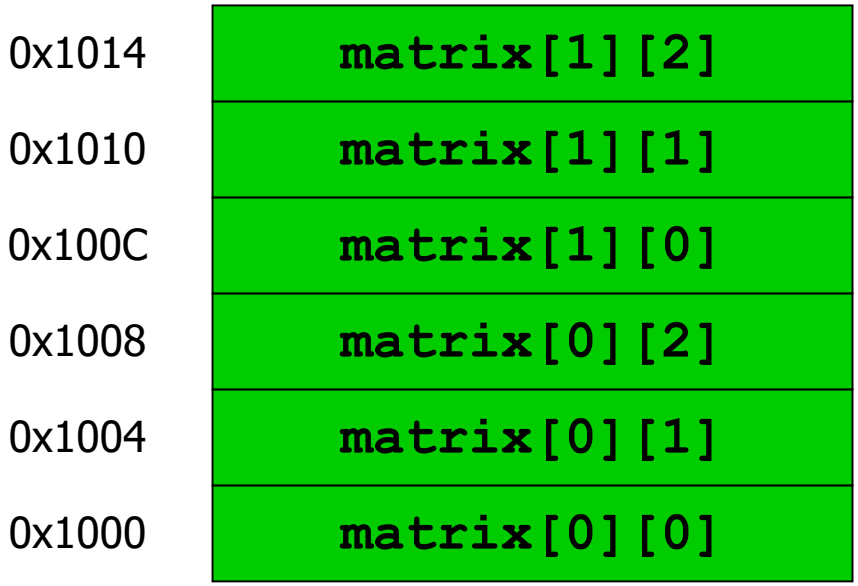
  int arr[2][2]={1,2,0,1};

  int arr[3][3]={{1,2,0},{1,4,6},{3,8,3}};

# Two Dimensional Arrays

- Array elements are stored contiguously one after the other.

```
int  matrix[2][3];

matrix[1][0] = 17;
```



# Two Dimensional Arrays

- It is convenient to think of a two-dimentional array as a rectangular collection of elements with rows and columns.
- For example: int a[3][5]; We can think of the array elements arranged as follows

| | col 1 | col 2 | col 3 | col 4 | col 5 |
|---|---|---|---|---|---|
| row 1 | a[0][0] | a[0][1] | a[0][2] | a[0][3] | a[0][4] |
| row 2 | a[1][0] | a[1][1] | a[1][2] | a[1][3] | a[1][4] |
| row 3 | a[2][0] | a[2][1] | a[2][2] | a[2][3] | a[2][4] |

# Two Dimensional Arrays

- Example:

```
#include <stdio.h>

#define   M   3       /* number of rows */
#define   N   4       /* number of columns */

int main(void)
{
   int   a[M][N], i, j, sum = 0;

   for (i = 0; i < M; ++i)          /* fill the array */
      for (j = 0; j < N; ++j)
         a[i][j] = i + j;
   for (i = 0; i < M; ++i) {        /* print array values */
      for (j = 0; j < N; ++j)
         printf("a[%d][%d] = %d     ", i, j, a[i][j]);
      printf("\n");
   }
   for (i = 0; i < M; ++i)          /* sum the array */
      for (j = 0; j < N; ++j)
         sum += a[i][j];
   printf("\nsum = %d\n\n", sum);
   return 0;
}
```

# Two Dimensional Arrays

- Output:

```
a[0][0] = 0    a[0][1] = 1    a[0][2] = 2    a[0][3] = 3
a[1][0] = 1    a[1][1] = 2    a[1][2] = 3    a[1][3] = 4
a[2][0] = 2    a[2][1] = 3    a[2][2] = 4    a[2][3] = 5

sum = 30
```

# Example

```
#include <stdio.h>
void main()
{
  int arr1[2][2],brr1[2][2],crr1[2][2],i,j,n=2;

for(i=0;i<n;i++)   /*fill elements in the first matrix */
      {
         for(j=0;j<n;j++)
            scanf("%d",&arr1[i][j]);

      }
 for(i=0;i<n;i++)      /*fill elements in the second matrix */
      {
         for(j=0;j<n;j++)
             scanf("%d",&brr1[i][j]);
       }
 printf("\nThe First matrix is :\n");
for(i=0;i<n;i++)
   {
     printf("\n");
     for(j=0;j<n;j++)
      printf("%d\t",arr1[i][j]);
   }

  printf("\nThe Second matrix is :\n");
  for(i=0;i<n;i++)
   {
     printf("\n");
     for(j=0;j<n;j++)
     printf("%d\t",brr1[i][j]);
   }
for(i=0;i<n;i++)
      for(j=0;j<n;j++)
          crr1[i][j]=arr1[i][j]+brr1[i][j];
  printf("\nThe Addition of two matrix is : \n");
  for(i=0;i<n;i++){
      printf("\n");
      for(j=0;j<n;j++)
          printf("%d\t",crr1[i][j]);
  }
  printf("\n\n");
}
```

```c
#include <stdio.h>
void main()
{
  int arr1[2][2],brr1[2][2],crr1[2][2],i,j,n=2;

for(i=0;i<n;i++)   /*fill elements in the first matrix */
      {
         for(j=0;j<n;j++)
            scanf("%d",&arr1[i][j]);

      }
for(i=0;i<n;i++)      /*fill elements in the second matrix */
      {
         for(j=0;j<n;j++)
             scanf("%d",&brr1[i][j]);
       }
 printf("\nThe First matrix is :\n");
for(i=0;i<n;i++)
   {
     printf("\n");
     for(j=0;j<n;j++)
      printf("%d\t",arr1[i][j]);
   }

  printf("\nThe Second matrix is :\n");
  for(i=0;i<n;i++)
   {
     printf("\n");
     for(j=0;j<n;j++)
     printf("%d\t",brr1[i][j]);
   }
for(i=0;i<n;i++)
      for(j=0;j<n;j++)
          crr1[i][j]=arr1[i][j]+brr1[i][j];
  printf("\nThe Addition of two matrix is : \n");
  for(i=0;i<n;i++){
      printf("\n");
      for(j=0;j<n;j++)
          printf("%d\t",crr1[i][j]);
  }
  printf("\n\n");
}
```

# Output

```
The First matrix is :
1       2
3       4
The Second matrix is :
5       6
7       8
The Addition of two matrix is :
6       8
10      12
```

# Dynamic Memory Allocation

# Dynamic Memory Allocation (DMA)

- Having an array size given by a specific constant in a program.

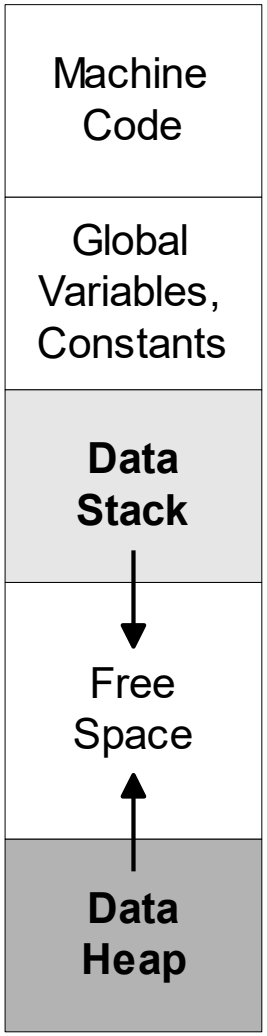
```
/*Fill and Print an array. */
#include <stdio.h>

#define   N   5

int main(void)
{
   int   a[N];      /* allocate space for a[0] to a[4] */
   int   i, sum = 0;

   for (i = 0; i < N; ++i)          /* fill the array */
      a[i] = 7 + i * i;
   for (i = 0; i < N; ++i)          /* print the array */
      printf("a[%d] = %d    ", i, a[i]);
   for (i = 0; i < N; ++i)          /* sum the elements */
      sum += a[i];
   printf("\nsum = %d\n", sum);     /* print the sum */
   return 0;
}
```

- Dynamic memory allocation: It may be desirable to allow the user to input the array size or to obtain the array size in a computation. We use some function calls to do the DMA

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
int *a, i, n, sum;
printf("%s", "input an array size n ");
scanf("%d", &n);
a=calloc(n, sizeof(int))  /* get space for n ints */
```

# Program Code and Space

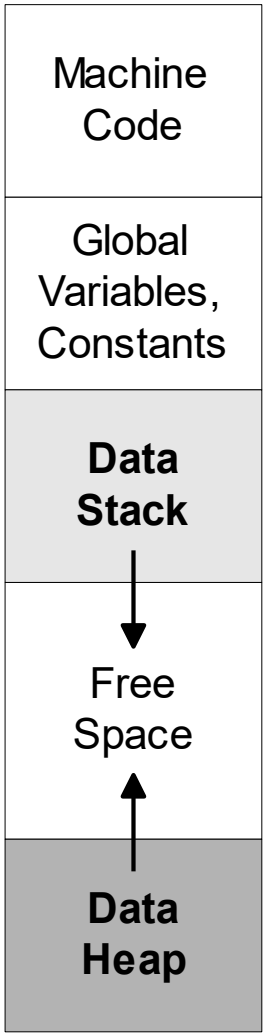
- Space for program code includes space for machine language code and data
- Data can be broken into:
  - Space for global variables and constants
  - Data Stack: it can expands/shrinks while program runs;
  - Data Heap: it can expands/shrinks while program runs



# The Stack

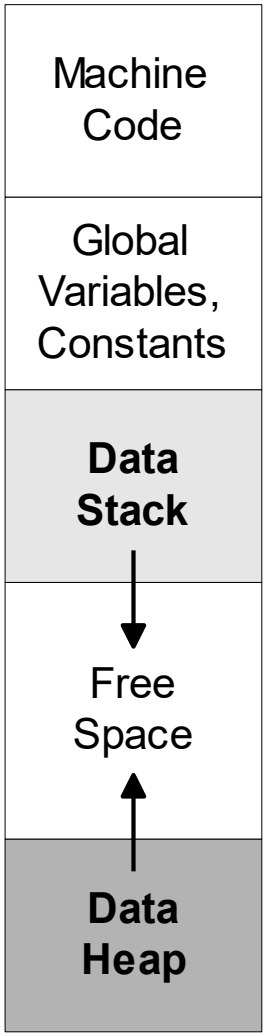
- The stack is the place where all local variables are stored.
  - A local variable is declared in some scope, space put aside on the data stack
  - Example:

    int x; //creates the variable x on the stack
  - As soon as the scope ends, all local variables declared in that scope end. That is, (1) the variable name and its space are gone; (2) this happens implicitly, that is, the user has no control over it.



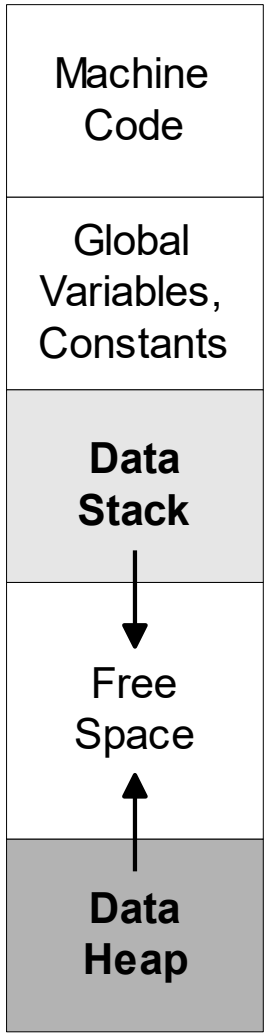
# The Heap

- The heap is an area of memory that the user handles explicitly.
  - User requests and releases the memory through system calls
  - If a user forgets to release memory, it does not get destroyed. It just uses up extra memory
- A user maintains a handle on memory allocated in the heap with a pointer



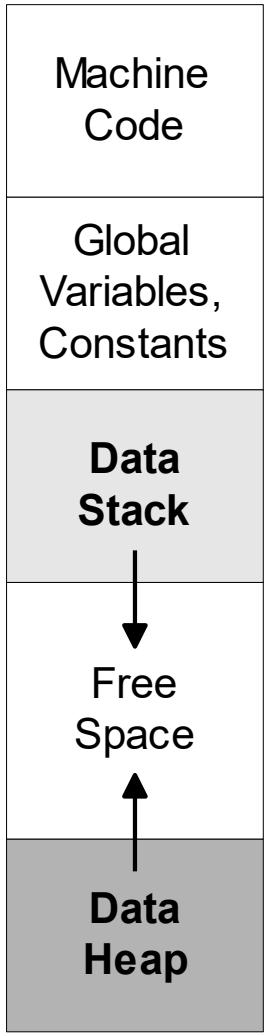
# The Stack: Static Memory Allocation

- The stack is the place where all local variables are stored.
- Local variables in functions are allocated when function starts; When function ends, space is freed up.
- Must know size of data item (int, array, etc.) when allocated (static allocation)



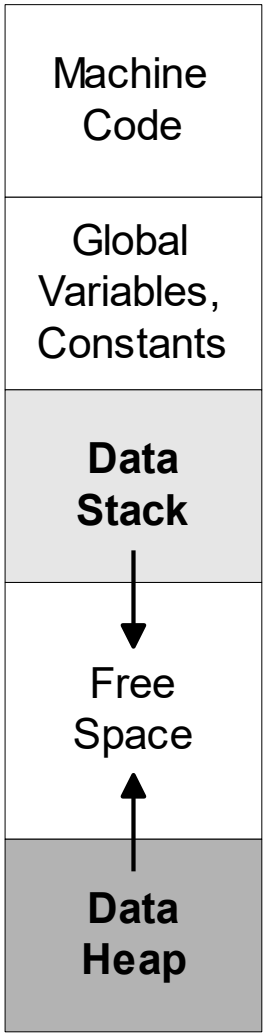
# Dynamic Memory Allocation

- Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)
- Previous example: ask the user how many numbers to read, then allocate array of appropriate size
- Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done
  - memory allocated in the *Data Heap*



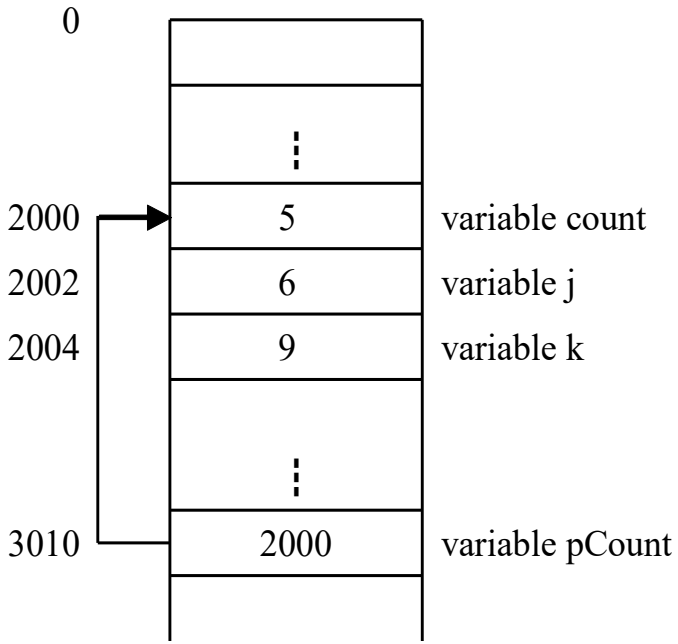
# The Heap

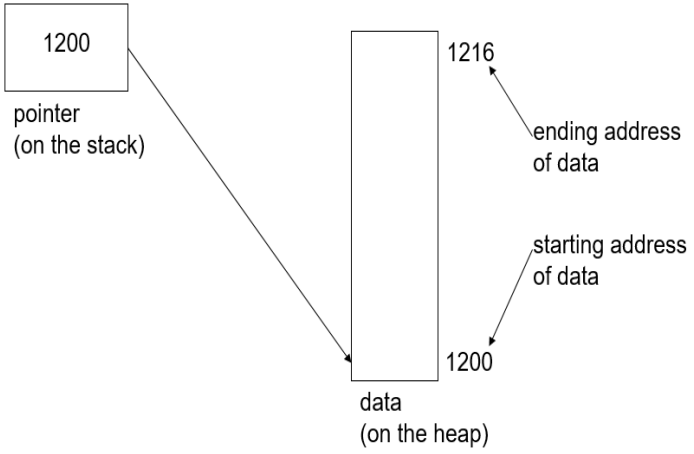
- The heap is an area of memory that the user handles explicitly.
  - User requests and releases the memory through system calls
  - If a user forgets to release memory, it does not get destroyed. It just users up extra memory
- A user maintains a handle on memory allocated in the heap with a pointer



# Pointer Declaration

- Declare a pointer: int *x;
- Using this pointer now would be very careful since x points to some random piece of data.
- Declaring a pointer variable does not allocate space for it.
  - It simply creates a local pointer variable (on the stack).
- Assign the address to a pointer, to let it point to a location on the stack.
- Use the dynamic memory allocation, we can let a pointer refer/point to a memory location on the heap; and Accessing the pointer, actually references the memory on the heap
  - For example, Use malloc() to actually request memory on the heap.



# sizeof() Function

- The *sizeof()* function is used to determine the size of any data type
- prototype: *int sizeof(data type);*
- Returns how many bytes the data type needs
- Example
  - sizeof(int) = 4, sizeof(char) = 1

# Memory Management Functions

- Programing C provides the following functions in the standard library for dynamic memory allocation by including stdlib.h head file.
  - calloc(): contiguous allocation. It is used to allocate arrays of memory
  - malloc(): memory allocation. It is used to allocate a single block of memory
  - realloc(): reallocation. It is used to change the amount of space allocated previously;
  - free(): It is a routine used to tell program a piece of memory no longer needed

- Noted that memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up.

# Function calloc()

- The form of function call
  - calloc(n, object_size)
  - n is the number of elements to be allocated
  - object_size is the size of a special type element to be allocated. We generally use sizeof(datatype) to get correct value.
  - An amount memory of size n*object_size will be allocated on heap.
- calloc returns the address of the first byte of this allocated memory
- if not enough memory is available, calloc returns NULL

# calloc Example

```
float *nums;
int N;
int I;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));
/* nums is now an array of floats of size N */
for (I = 0; I < N; I++) {
  printf("Please enter number %d: ",I+1);
  scanf("%f",&(nums[I]));
}
/* Calculate average, etc. */
```

# Fucntion malloc()

- The form of function call:
  - malloc(esize)
    - esize is the size of total number of special type elements to be allocated. We generally use the number of element and the sizeof(datatype) to get the correct value.
- As with calloc, memory is allocated from heap
- NULL returned if not enough memory available
- It can perform the same function as calloc if we simply multiply the two arguments of calloc together.
- Example

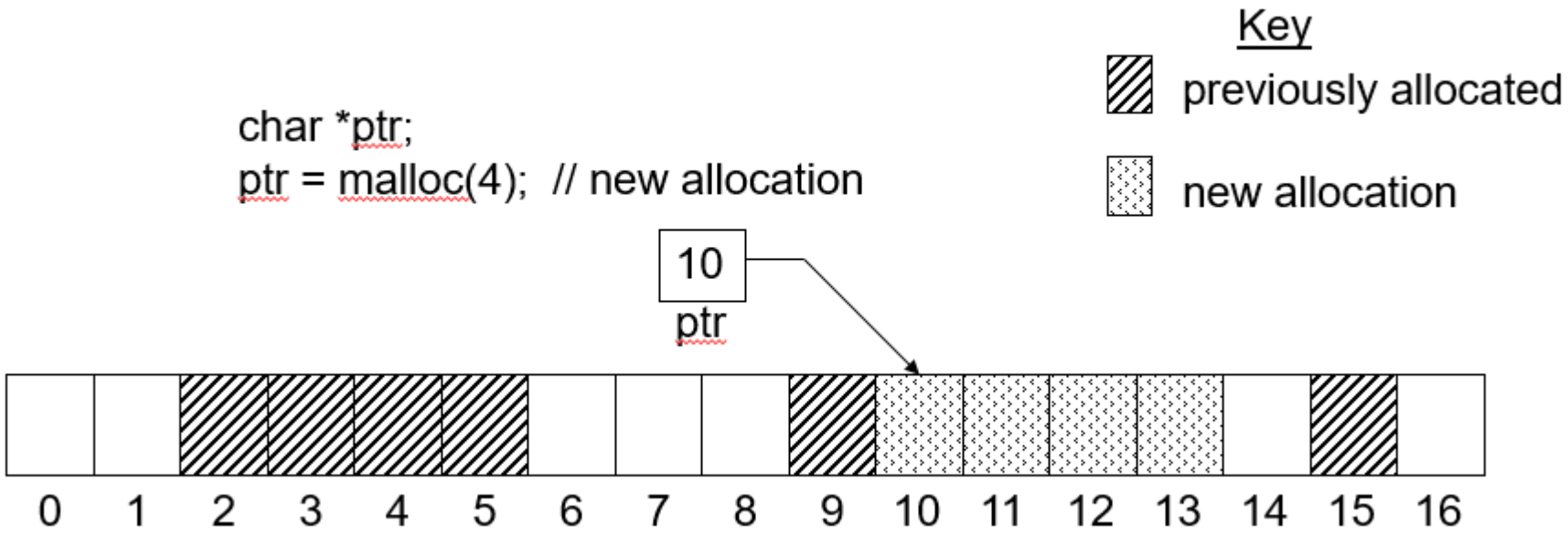
  malloc(N * sizeof(float)) is equivalent to

  calloc(N, sizeof(float))

# malloc: Array Allocation with malloc

- Example: malloc (esize);
  - function searches heap for *size* contiguous free bytes
  - function returns the address of the first byte

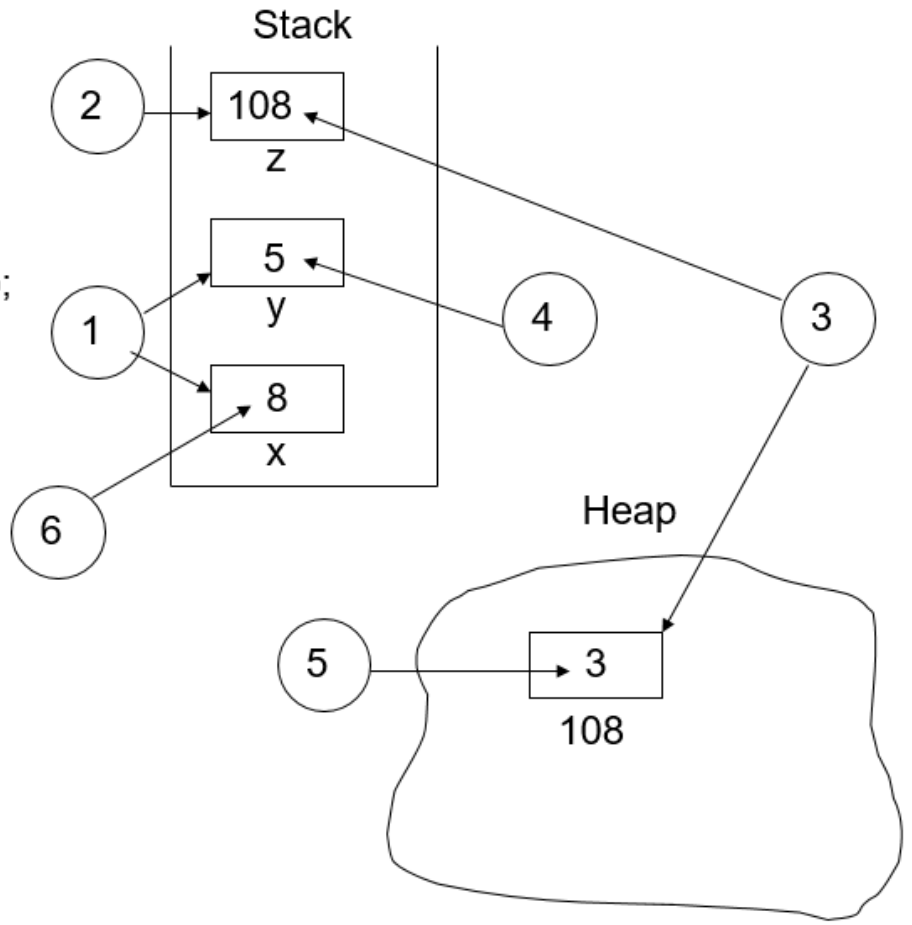
- Example



# malloc: Array Allocation with malloc

- Example

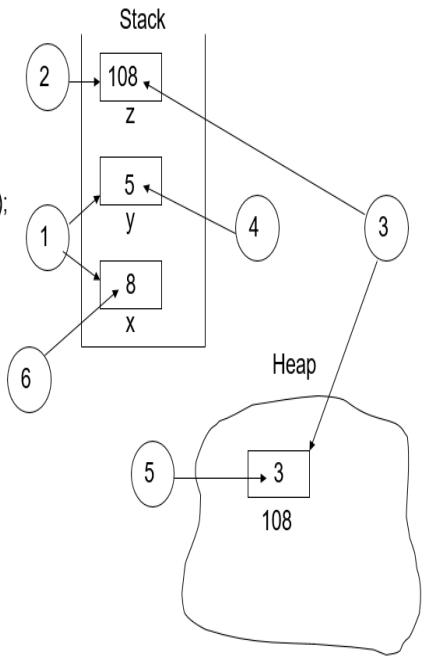
```
   int main() {
1      int x, y;
2      int *z;
3      z = malloc(sizeof(int));

4      y = 5;
5      *z = 3;
6      x = *z + y;
7      free(z);

       return 0;
   }
```



# malloc: Array Allocation with malloc

1. Declare local variables x and y.
2. Declare local pointer z.
3. Allocate space on the heap for single integer. This step also makes z point to that location (notice the address of the space on the heap is stored in z's location on the stack.
4. Set the local variable y equal to 5.
5. Follow the pointer referenced by z to the heap and set that location equal to 3.
6. Grab the value stored in the local variable y and follow the pointer z to grab the value stored in the heap. Add these two together and store the result in the local variable x.
7. Releases the memory on the heap (so another process can use it) and sets the value in the z pointer variable equal to NULL. (this step is not shown on the diagram)

```
   int main() {
1     int x, y;
2     int *z;
3     z = malloc(sizeof(int));

4     y = 5;
5     *z = 3;
6     x = *z + y;
7     free(z);

      return 0;
   }
```



# Function realloc()

- The form of function call
  - realloc(ptr, new_size)
  - ptr is a pointer we dynamically allocated the space to it previously
  - new_size is the size we update.
  - Extend/change the amount of space allocated previously;.
- Change memory size that is already allocated dynamically to a variable.

# Function realloc()

- Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *ptr = (int *)malloc(sizeof(int)*2);
int i;
int *ptr_new;
*ptr = 10;
*(ptr + 1) = 20;
ptr_new = (int *)realloc(ptr, sizeof(int)*3);
*(ptr_new + 2) = 30;
for(i = 0; i < 3; i++)
printf("%d ", *(ptr_new + i));
return 0;
}
```

- Output

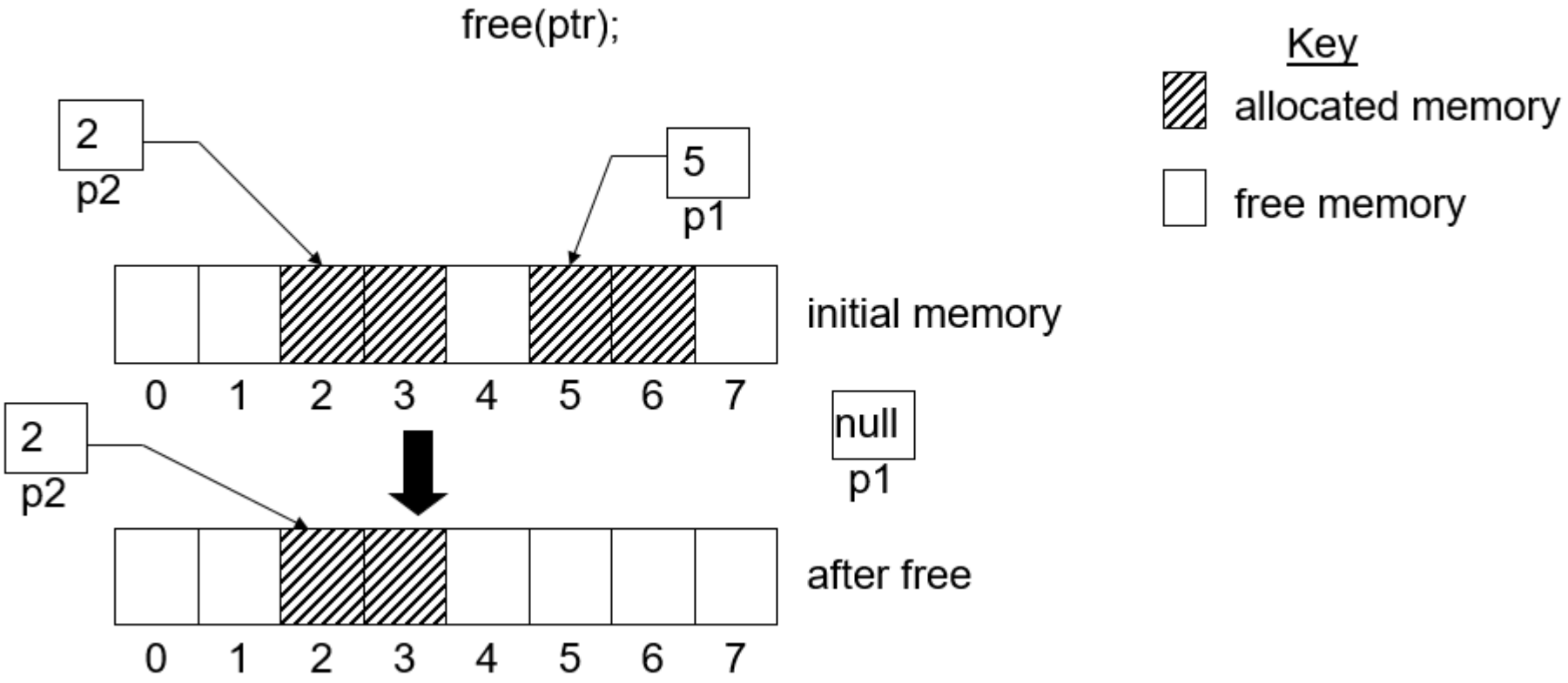
  10 20 30

# free: Releasing memory with free

- The form of function call
  - void free (void *ptr)
  - memory at location pointed to by ptr is released (so we could use it again in the future)
- program keeps track of each piece of memory allocated by where that memory starts
- if we free a piece of memory allocated with calloc, the entire array is freed (released)

# free: Releasing memory with free

- Example: int free (int ptr)
  - releases the area pointed to by ptr
  - ptr must not be null



# free Example

```
float *nums;
int N;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));

/* use array nums */

/* when done with nums: */

free(nums);

/* would be an error to say it again - free(nums) */
```