Dynamic Memory Allocation

Motivation

- We want to read in a dictionary of words
- Before reading it in:
  - We don't know how many words there are
  - We don't know how big each word is

Possible Solution

- Allocate the maximum amount you could ever need
- Question: why is this generally not a good solution? (2 reasons)

```
// 1000 words max with
// 100 characters max per word
char dictionary[1000][100];
```

Problems

- Most things do not have a good "maximum"
- Your program always needs the maximum amount of memory, and usually the vast majority is completely wasted

What is Desired

- A way to tell the computer to give a certain amount of memory to a program as it runs
- Only what is explicitly requested is allocated

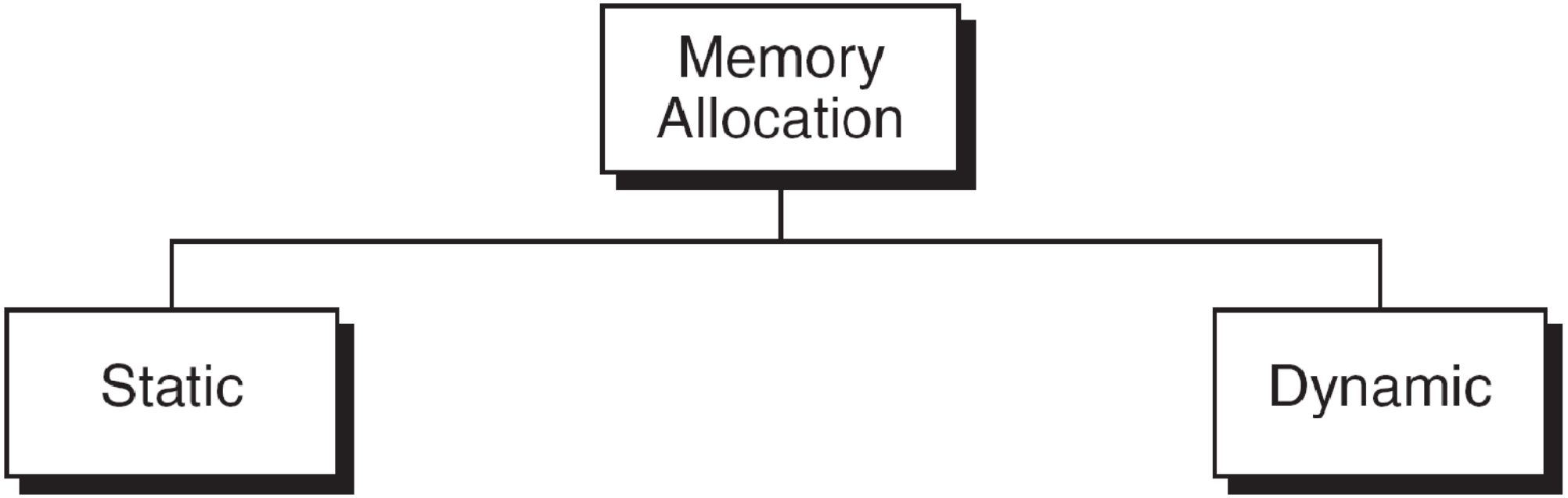
Dynamic Memory Allocation

- Dynamic: as the program runs
- Memory allocation: set aside memory

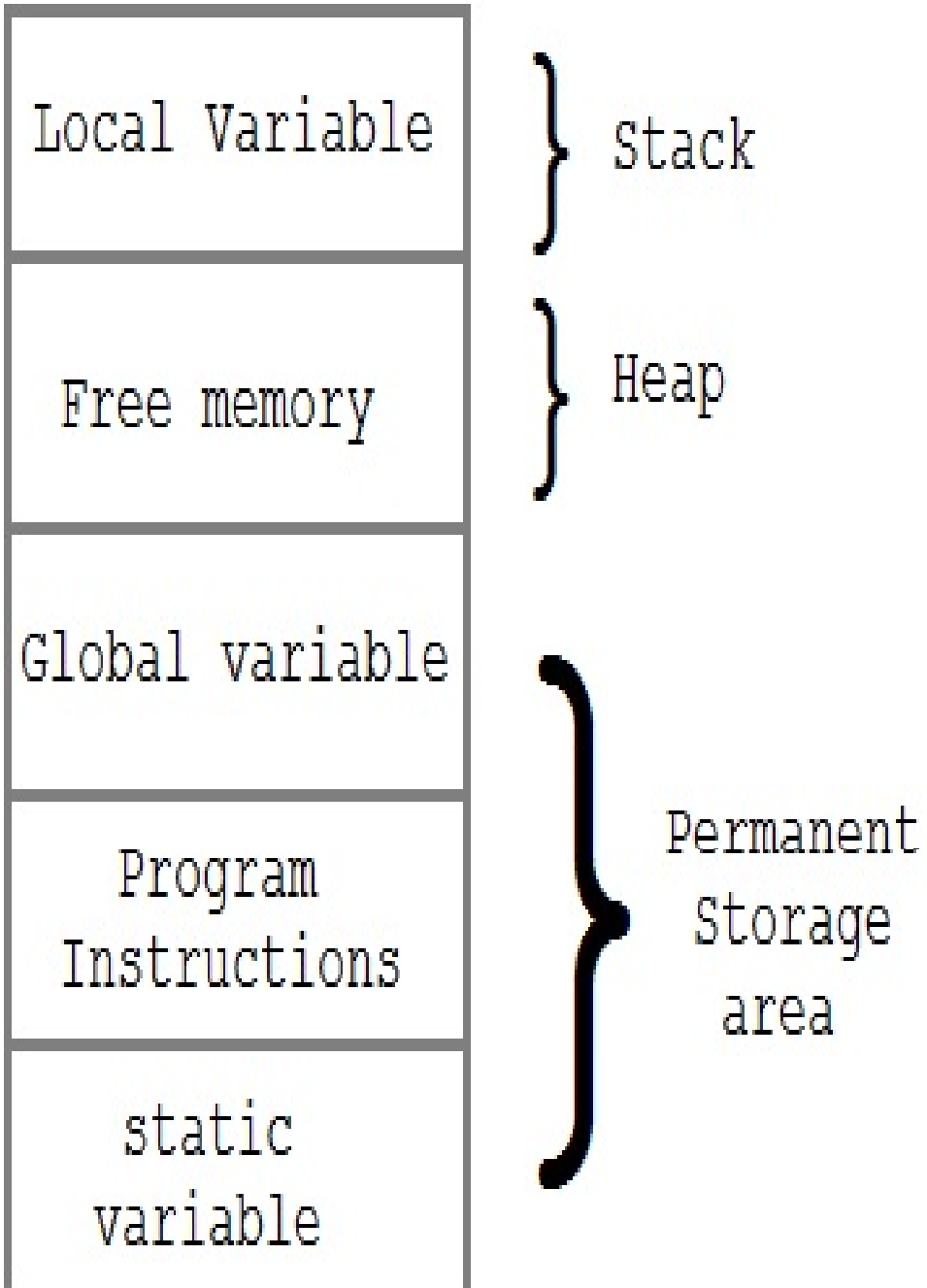
Introduction

C has two choices to reserve memory locations for an object:

- Static allocation
- Dynamic allocation.

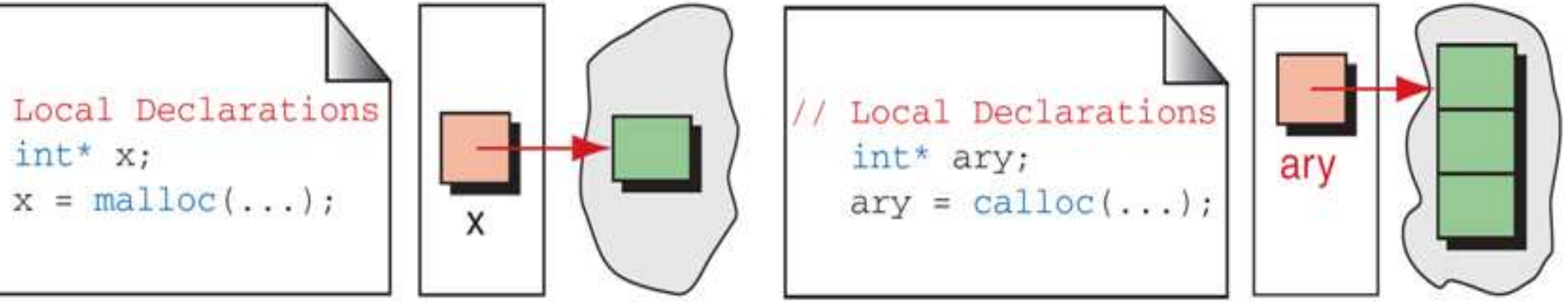


A Conceptual View of Memory



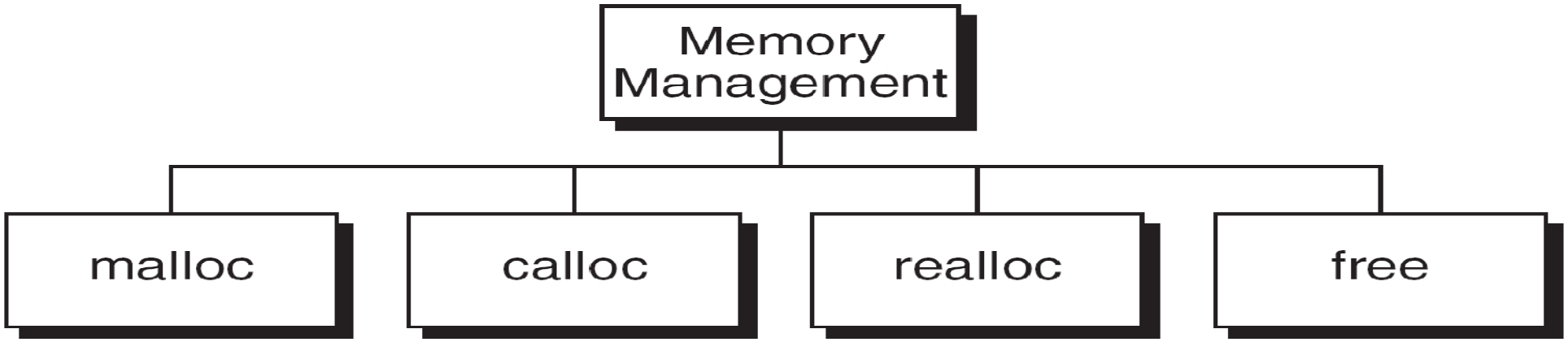
Accessing Dynamic Memory

(a) Static Memory Allocation



(b) Dynamic Memory Allocation

Memory Management Functions



Function	Description
malloc	Allocates the specified number of bytes
ealloc	Increases or decreases the size of the specified block of memory. Reallocates it if needed
alloc	Allocates the specified number of bytes and initializes them to zero
ree	Releases the specified block of memory back to the system

malloc

- The most generic way to allocate memory
- Takes the number of bytes to allocate
- Returns a `void*` to the block of memory allocated

```
void* malloc( int numBytes );
```

Using malloc

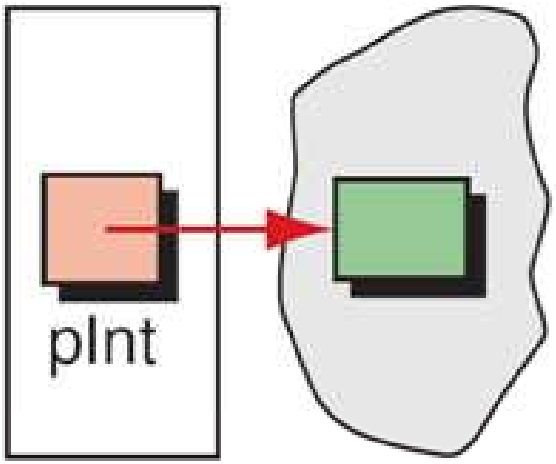
- The `sizeof` operator comes in handy
  - Returns an integral size
- For example: allocate room for 50 integers dynamically:

```
// dynamically
int* nums1;
nums1 = malloc( sizeof( int ) * 50 );

int nums2[ 50 ]; // statically
```

malloc

- The most generic way to allocate memory
- Takes the number of bytes to allocate

```
pInt = malloc(sizeof(int))
```



calloc

- Very similar to malloc
- Takes the number of elements to allocate and the size of each element
  - Will do the multiplication itself
- Will also initialize the allocated portion to zero

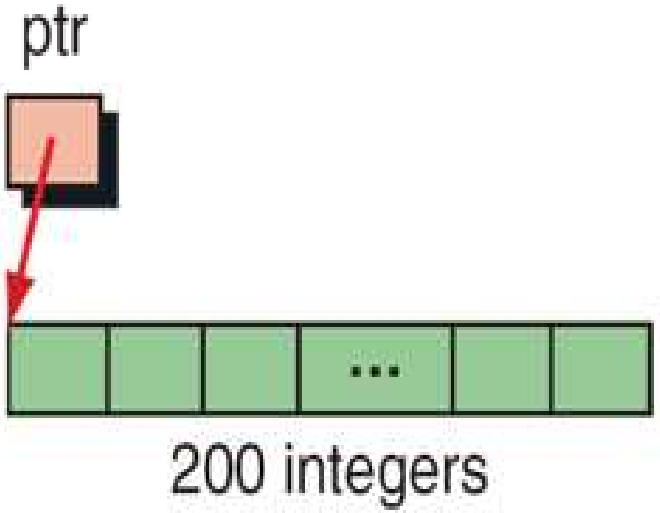
```
void* calloc( int num, int size );
```

calloc

- Very similar to malloc

```
int* nums1, nums2;
nums1 = malloc( sizeof( int ) * 50 );
nums2 = calloc( 50, sizeof( int ) );
```

calloc




```
ptr = (int*)calloc (200, sizeof(int))
```

realloc

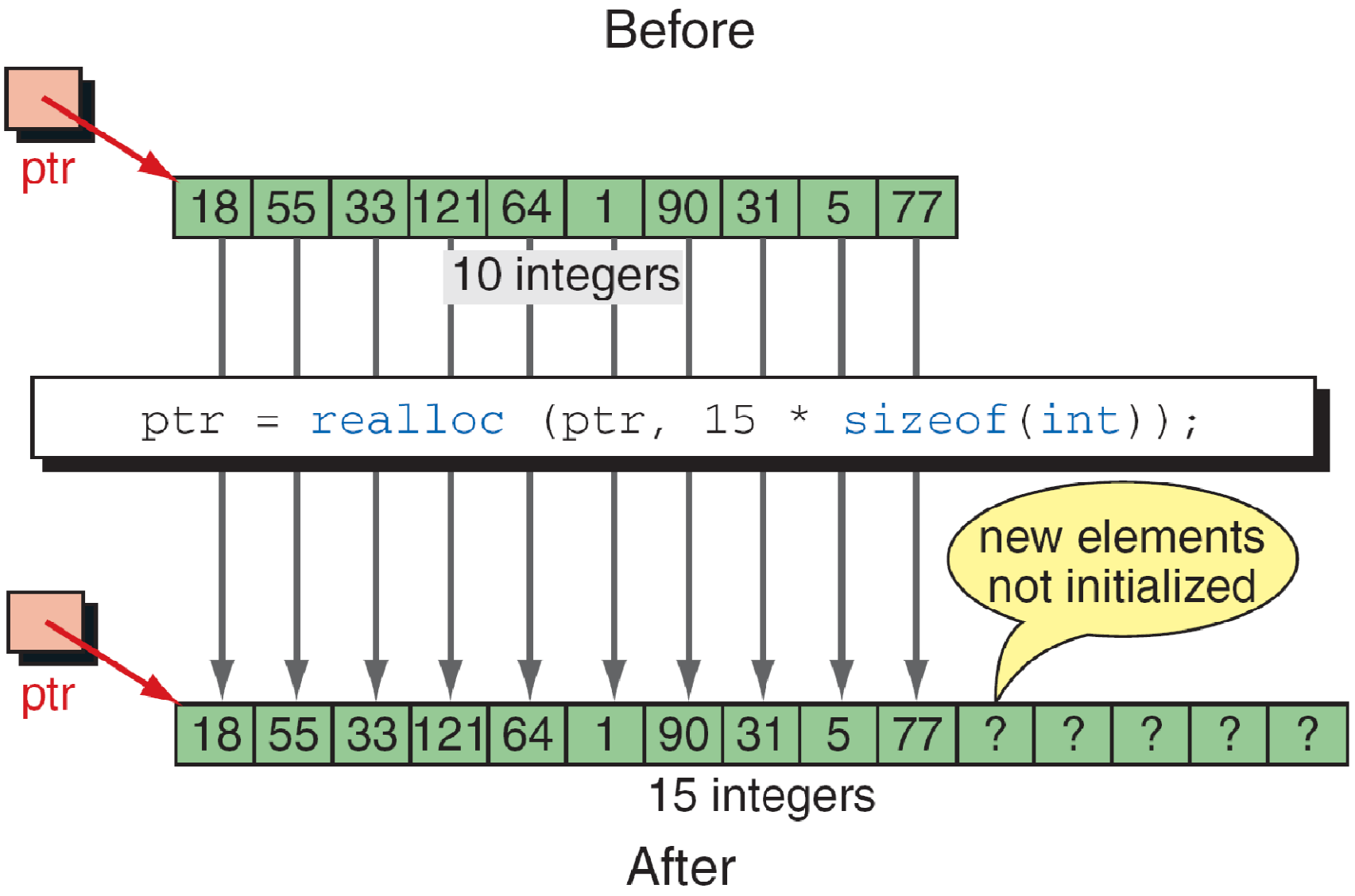
- For resizing a block of memory that has already been allocated (with one of `malloc`, `calloc`, or `realloc`)
  - Except if given `NULL` - then it behaves like `malloc`
- Takes the previously allocated memory block, and the new size

```
void* realloc( void* ptr, int size );
```

realloc

```
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// we want 5 more integers
nums = realloc( nums,
                sizeof( int ) * 55 );
```

realloc

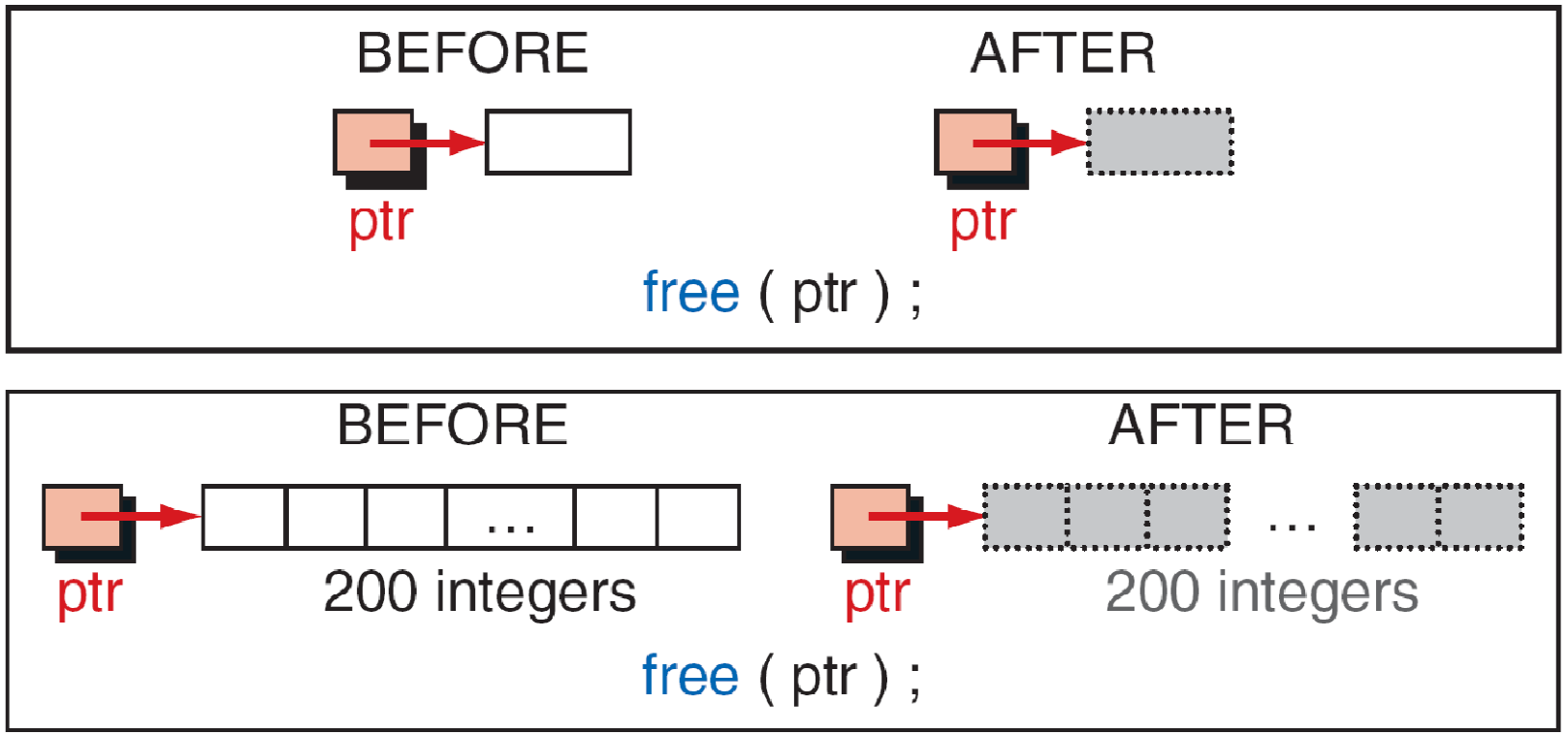

Before
ptr
18 55 33 121 64 1 90 31 5 77
10 integers
ptr = realloc (ptr, 15 * sizeof(int));
new elements
not initialized
ptr
18 55 33 121 64 1 90 31 5 77 ? ? ? ? ?
15 integers
After

```
#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;
}
```

- Once we are done using a block of memory, call free on it
- If a block is never freed, it is called a **memory leak**
  - Memory is still allocated but wasted

```
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// done with nums
free( nums );
```

free


BEFORE
AFTER
ptr
ptr
free (ptr) ;
BEFORE
AFTER
ptr
200 integers
ptr
200 integers
free (ptr) ;

sing a single pointer:

A simple way is to allocate memory block of size r*c and access elements using simp
pointer arithmetic.

```
int *arr = (int *)malloc(r * c * sizeof(int));
```

Eg. scanf("%d",&*(arr + i*c + j));

sing an array of pointers

After creating an array of pointers, we can dynamically allocate memory for every r

```
int *arr[r];
   for (i=0; i<r; i++)
            arr[i] = (int *)malloc(c * sizeof(int));
```

sing pointer to a pointer

Once we have an array pointers allocated dynamically, we can dynamically allocate
memory and for every row

```
int **arr = (int **)malloc(r * sizeof(int*));
  for (i=0; i<r; i++)
```

```
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");   exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
```