Dynamic Allocation using Pointers in C and C++

Dynamic Memory Management

- You can control the *allocation* and *deallocation* of memory in a program for objects and for arrays of any built-in or user-defined type.
  - Known as dynamic memory management; performed with malloc and free (in C) and new and delete (in C++ or Java)
- You can use the new or malloc operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an array at execution time.
- The built-in array is created in the free store (also called the heap)—*a region of memory assigned to each program for storing dynamically allocated objects.*
- Once memory is allocated in the free store, you can access it via the pointer that operator new or malloc returns.
- You can return memory to the free store by using the delete or free operator to deallocate it.

Dynamic Memory Management (cont.)

Obtaining Dynamic Memory with new

- The malloc call allocates storage of the proper size and returns a **void** pointer. This pointer can be cast to the appropriate type and assigned to a variable.
- If malloc is unable to find sufficient space in memory, it returns NULL.

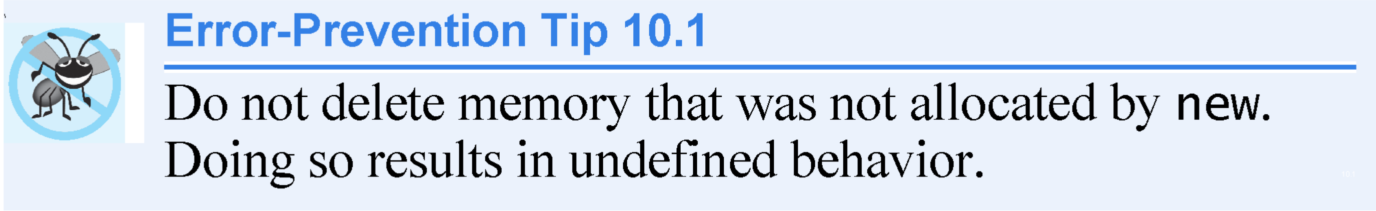
Dynamic Memory Management (cont.)

Releasing Dynamic Memory with free

- To destroy a dynamically allocated object, use the delete or free operator as follows:
  - free ptr;
- This statement first *deallocates the memory associated with the pointer, returning the memory to the free store.*

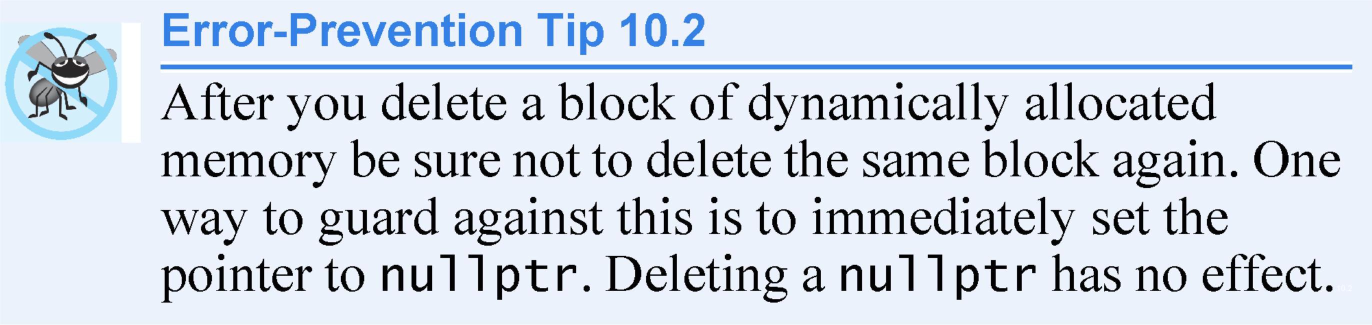
Common Programming Error 10.2

Not releasing dynamically allocated memory when it’s no longer needed can cause the system to run out of memory prematurely. This is sometimes called a “memory leak.”



Error-Prevention Tip 10.1

Do not delete memory that was not allocated by `new`. Doing so results in undefined behavior.



Error-Prevention Tip 10.2

After you delete a block of dynamically allocated memory be sure not to delete the same block again. One way to guard against this is to immediately set the pointer to `nullptr`. Deleting a `nullptr` has no effect.

Examples In C

Dynamic Memory Management

Initializing Dynamic Memory

- C provides several functions for dynamic memory management. These are mostly defined in <stdlib.h> library:

1. **void *calloc (int num, int size);**

   //This function allocates an array of **num** elements each of which size in bytes will be **size**.

2. **void free (void *address);**

   //This function releases a block of memory block specified by address.

3. **void *malloc (int num);**

   //This function allocates an array of **num** bytes and leave them uninitialized.

4. **void *realloc (void *address, int newsize);**

   //This function re-allocates memory extending it upto **newsize**.

Dynamic Allocation

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int num, *ptr;
  scanf (“%d”, &num);

 /* allocate memory dynamically */
  ptr = (int*) malloc( num * sizeof(int) );
  if( ptr == NULL )
    fprintf(stderr, "Error - unable to allocate required memory\n");

  free(ptr);
}
```

Allocating a Character Array

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  char name[100]; char *description;
  strcpy(name, “Harry Potter!");

  /* allocate memory dynamically */
  description = malloc( 200 * sizeof(char) );
  //description = calloc( 200, sizeof(char) );
  description = realloc( description, 100 * sizeof(char) );
  if( description == NULL )
    fprintf(stderr, "Error - unable to allocate required memory\n");
  else
    strcpy( description, “This is Demo for C");

  free(description);
  //printf("Name = %s \n Description: %s \n", name, description );
}
```

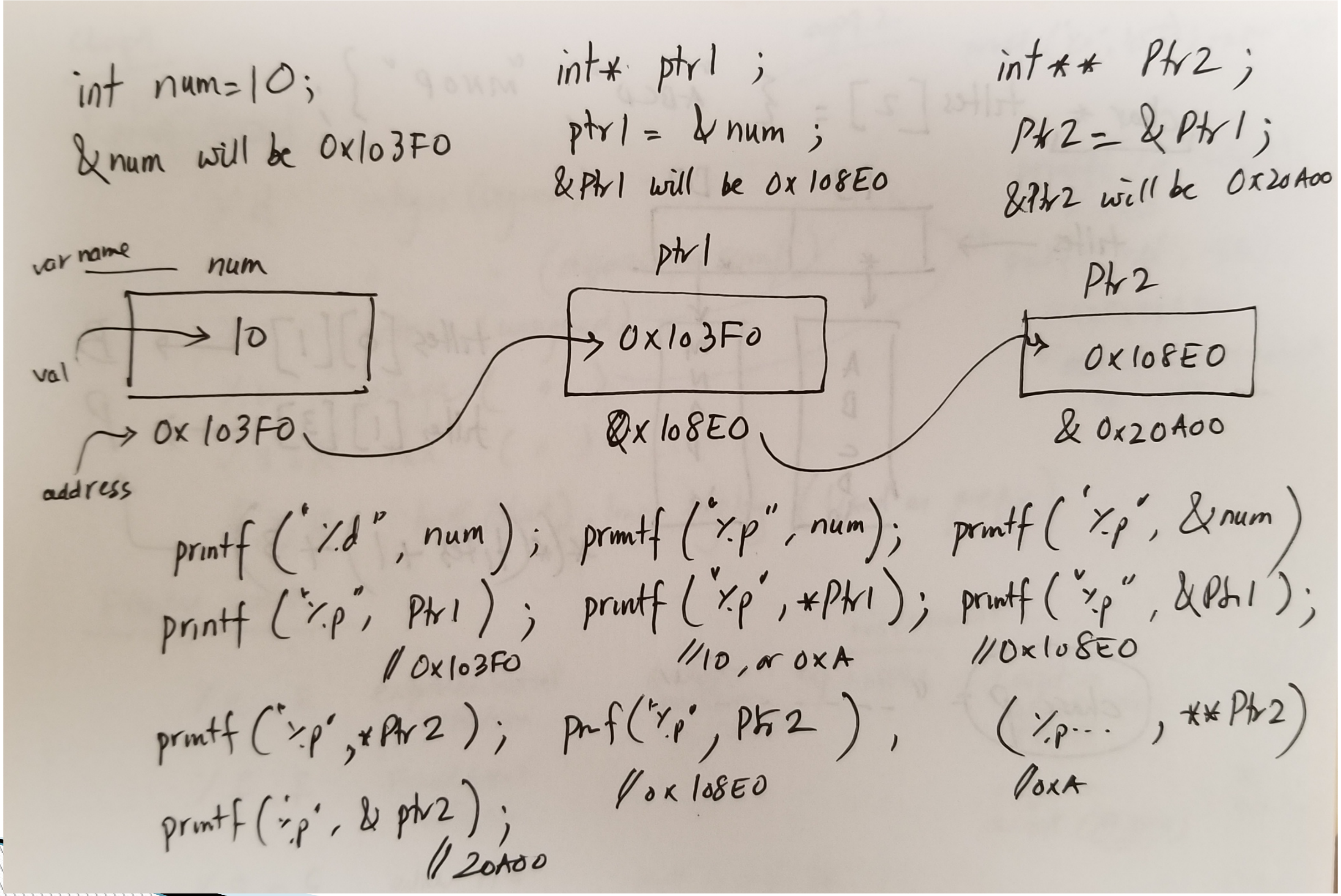
Allocating 2D Array as 1 malloc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
   int *d_array =  (int *) malloc( N * M * sizeof(int) );
   int *ptr = d_array;
   for(i=0; i < N; i++) {
     for(j=0; j < M; j++) {
        d_array[i*M +j] = 0;
        //*ptr = 0; ptr++;
     }
   }
   free(d_array);
}
```

Allocating 2D Array as 2 mallocs

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int N = 3, M = 5, i, j;
        int**d_array = (int**) malloc( N * sizeof(int*) );   //Allocating memory for 2D array
        for(i=0; i< N; i++)
            d_array[i] = (int*) malloc(M * sizeof(int) );
        for(i=0; i< N; i++) {                                //Initializing 2D array using [ ][ ] notation
            for(j=0; j < M; j++) {
                d_array[i][j] = i+j;
            }
        }
        for(i=0; i< N; i++) {                                 //Accessing 2D array using ** notation
            for(j=0; j < M; j++) {
                printf("%d ",*(*(d_array+i)+j));
            }
            printf("\n");
        }
        for(i=0; i< N; i++)                                   //Deallocating 2D array
                 free(d_array[i] );
        free(d_array);
}
```

Double Pointer



```
int num=10;
&num will be 0x103F0

int* ptr1 ;
ptr1 = &num ;
&Ptr1 will be 0x108E0

int** Ptr2 ;
Ptr2= &Ptr1;
&Ptr2 will be 0x20A00
```

```
printf ("%d", num);    printf ("%p", num);     printf ("%p", &num)
printf ("%p", Ptr1) ;  printf ("%p", *Ptr1);   printf ("%p", &Ptr1);
     // 0x103F0             //10, or 0xA            //0x108E0
printf ("%p",*Ptr2);   Prnf("%p", Ptr2) ,      (%p... , **Ptr2)
                            //0x108E0              //0xA
printf ("%p", & ptr2);
     // 20A00
```