Dynamic Memory Management

Tools for creating
dynamic data structures

Practical Data Structures

- We need *data structures* that can *grow* and *shrink* during execution.

- Implementing ***dynamic*** data structures needs two important aspects:
  - Dynamic memory allocations
  - Self-referential structures

Dynamic Memory Manipulations

- Dynamic Memory Allocation: **`malloc()`**

From the C reference manual:

```
/* header file stdlib.h declares the function */
#include <stdlib.h>

   void * malloc(size_t size);
```

- **`size_t`** has been type-defined to `unsigned int`.
- **`size`** is the number of bytes required in the allocation.
- **`malloc()`** returns a pointer to a block of memory of `size` bytes.
- The function call returns `NULL` when the allocation fails.

Get *some* Bytes Using `malloc()`

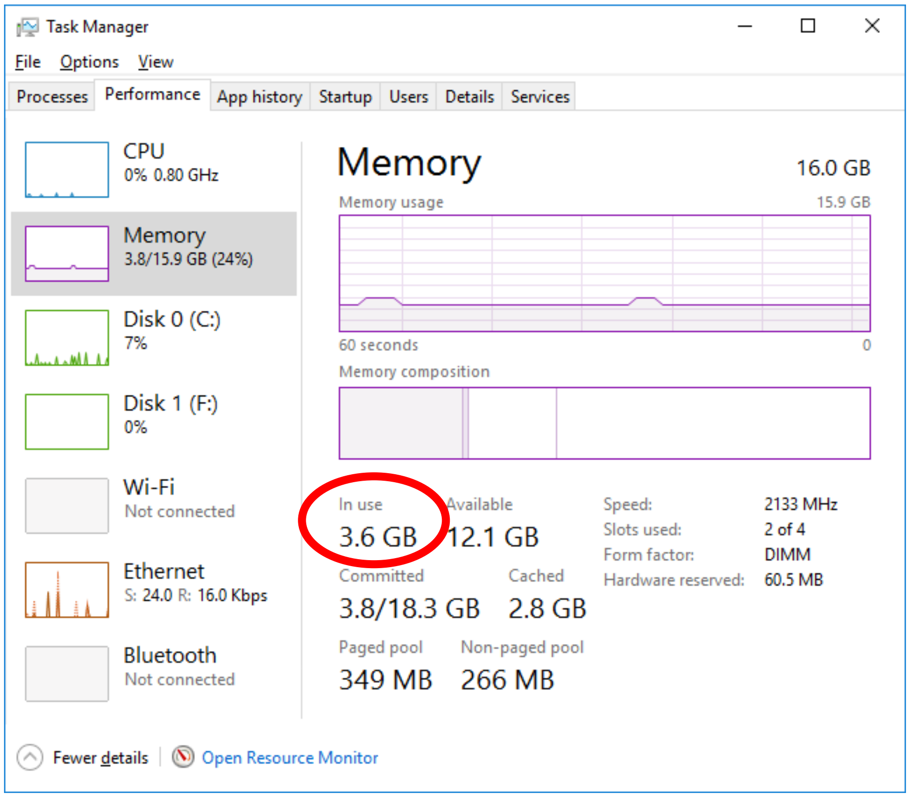
Program dyn_1.c

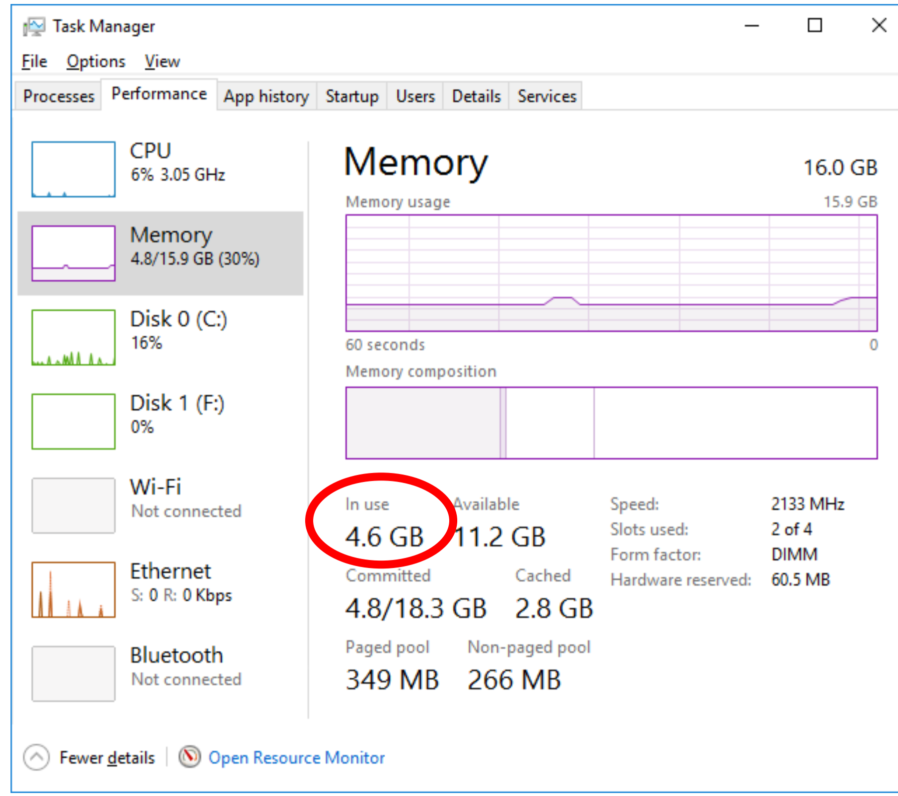
```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int amount;
  void *ptr;

  printf("How many bytes? ");
  scanf("%d", &amount);

  ptr = malloc(amount);

  if (ptr != NULL)
    printf("Address: %p\n", ptr);
  else
    printf("Failed to allocate memory!\n");

  return 0;
}
```

```
How many bytes? 1000000000↵
Address: fa265fd0
```

Get ~1GB of memory!

Memory Usage at a Glance!





Get *a few* Bytes Using `malloc()`

Program dyn_2.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int *ptr;                              /* Declare an integer pointer variable */

   ptr = (int *)malloc( sizeof(int) ); /* Call malloc() to get 4 bytes */

   if (ptr != NULL) {
     *ptr = 97;                /* Store an integer into the memory pointed by ptr */
     printf("The int stored at %p is %d.\n", ptr, *ptr);
   }

   return 0;
}
```

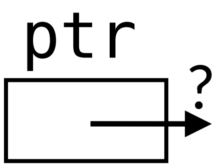
```
The int stored at fa265fd0 is 97.
```

Get the storage for an integer!

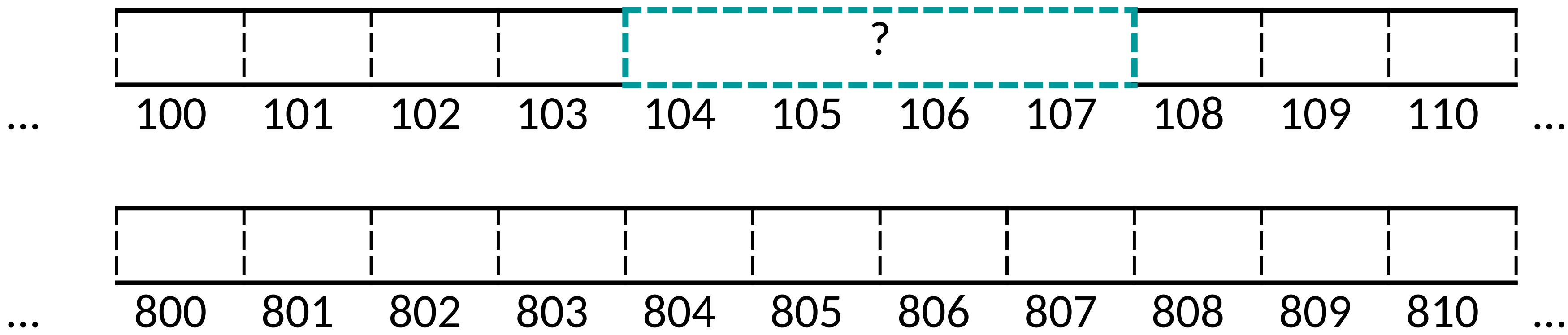
Memory allocation (using `malloc()`)

```
int *ptr;
ptr = (int
  *)malloc(sizeof(int));
*ptr = 97;
```

Pictorial View



Memory View

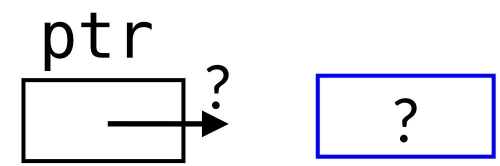


- **`ptr`**, a pointer variable, is automatically allocated a memory space for storing an address.

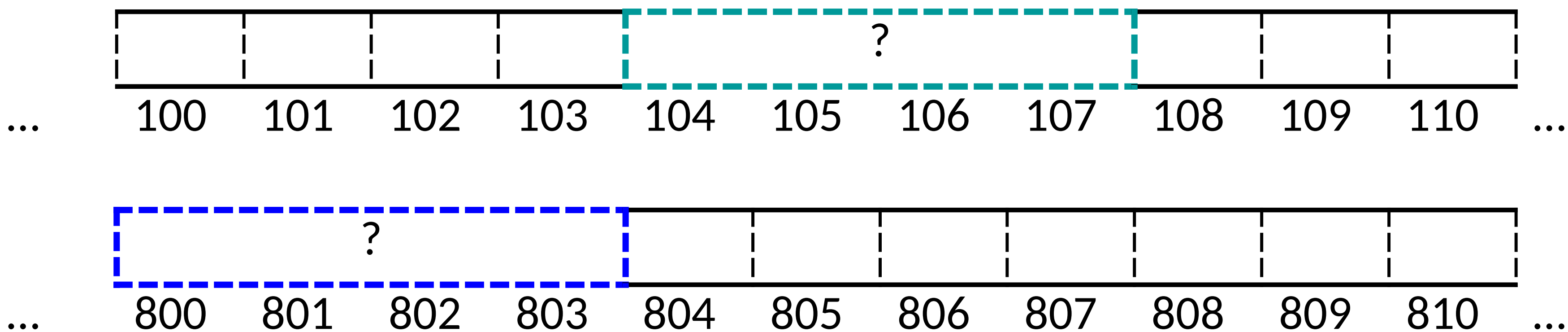
Memory allocation (using `malloc()`)

```c
int *ptr;
ptr = (int *)malloc(sizeof(int));
*ptr = 97;
```

Pictorial View



Memory View

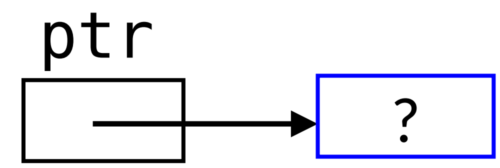


- `malloc()` reserves a memory space that is big enough to store a value of type `int`.

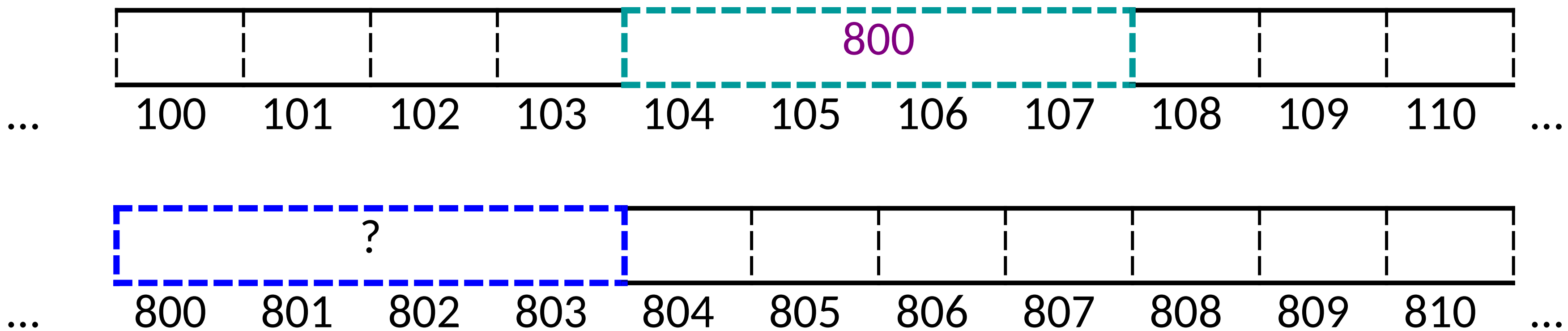
Memory allocation (using `malloc()`)

```
int *ptr;
ptr = (int *)malloc(sizeof(int));
*ptr = 97;
```

Pictorial View



Memory View

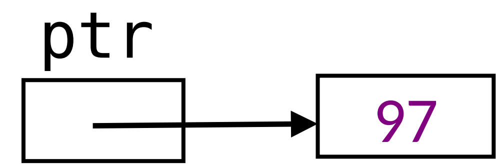


- `malloc()` returns the address of the reserved space and the address is assigned to **`ptr`**.

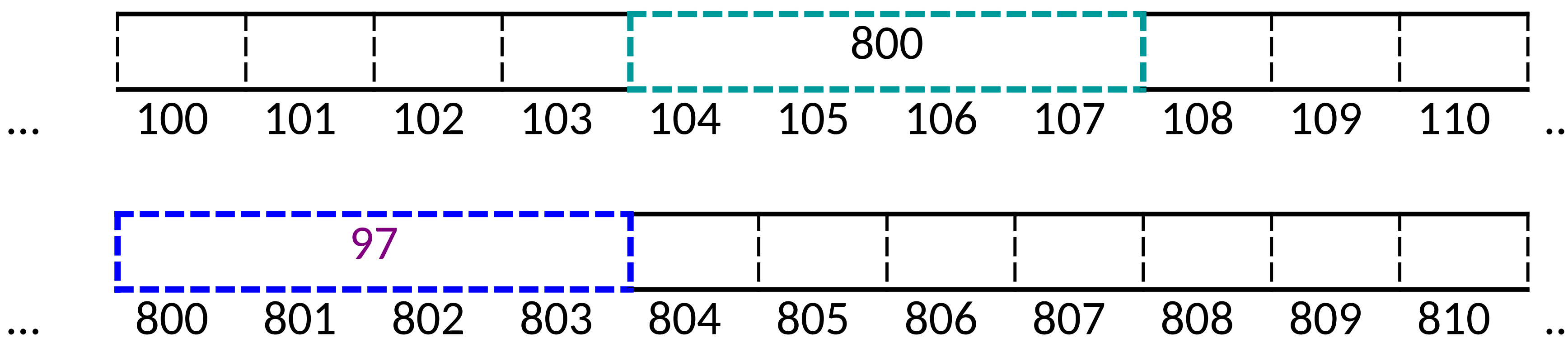
Memory allocation (using `malloc()`)

```
int *ptr;
ptr = (int
   *)malloc(sizeof(int));
*ptr = 97;
```

Pictorial View



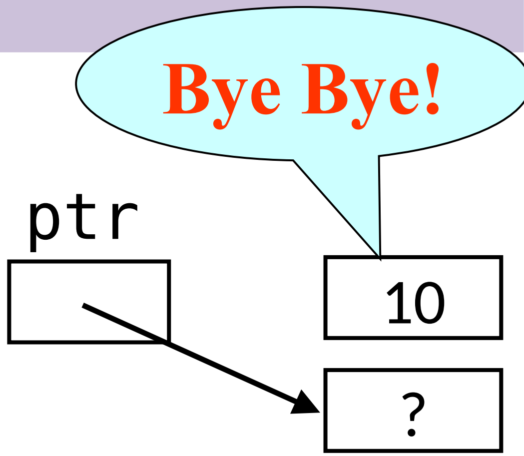
Memory View



- Store 97 in the allocated space pointed to by **`ptr`**.
- The allocated space is like a "variable of type `int`" except that it has no name.

Losing Allocated Space

```
int *ptr;
ptr = (int *)malloc(sizeof(int));
*ptr = 10;
ptr = (int *)malloc(sizeof(int));
```



- The allocated space has no corresponding name in the program.

- If you lose the memory address of an allocated space, you lose the data stored in that space and you lose the space.

- Allocated space will stay reserved until it is explicitly released or until the program terminates.

Dynamic Memory Manipulations

- Dynamic Memory De-Allocation: **`free()`**

From the C reference manual:

```
/* header file stdlib.h declares the function
*/
#include <stdlib.h>

void free(void * ptr);
```

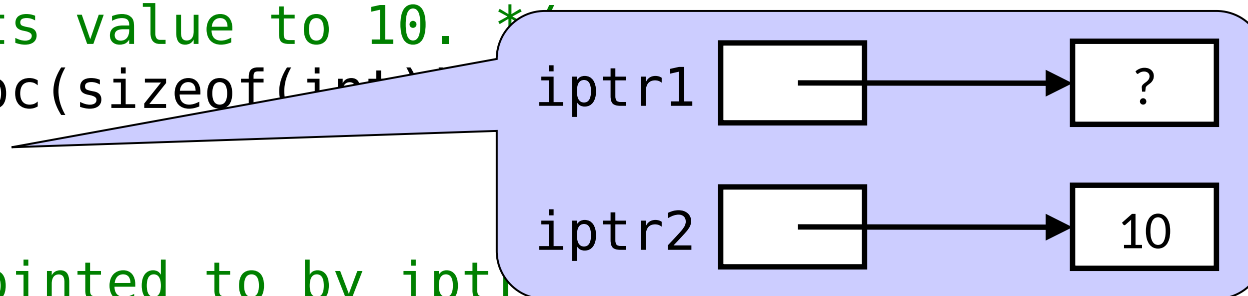
- **`void *`** is a *generic* pointer type, i.e. pointer of **ANY** type.
- **`ptr`** is a pointer to a block of memory *previously allocated* by `malloc()`.
- It releases the block of memory pointed to by **`ptr`**.

Complete: `malloc()`, then `free()`

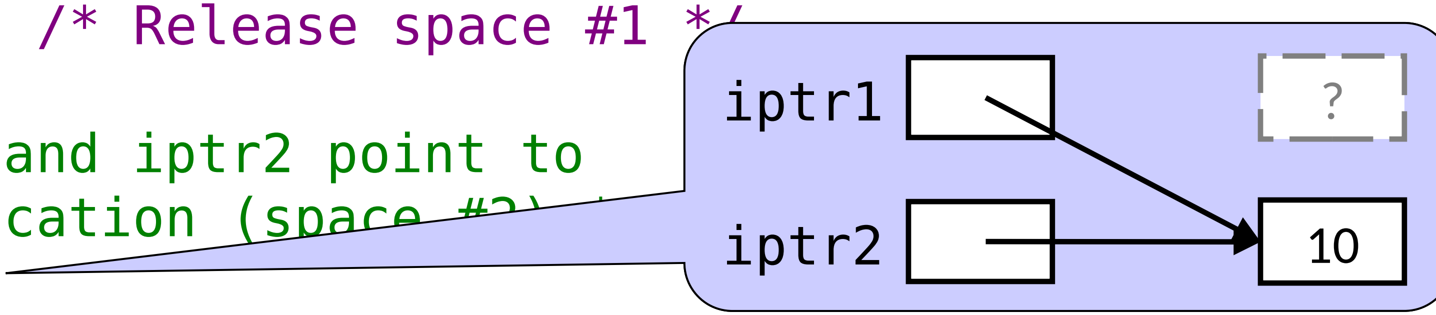
Program dyn_3.c

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int * ptr;      /* Declare an integer pointer variable, points to nothing initially */

  ptr = (int *)malloc(sizeof(int)); /* Call malloc() to get 4 bytes */
  if (ptr != NULL)
  {
    *ptr = 97;              /* Store an integer into the memory pointed by ptr */
    printf("The int stored at %p is %d.\n", ptr, *ptr);
  }

  free(ptr);          /* RELEASE the memory pointed by ptr, integer 97 is LOST */
  ptr = NULL;               /* SET ptr to NULL after free() as a good practice */

  return 0;
}
```

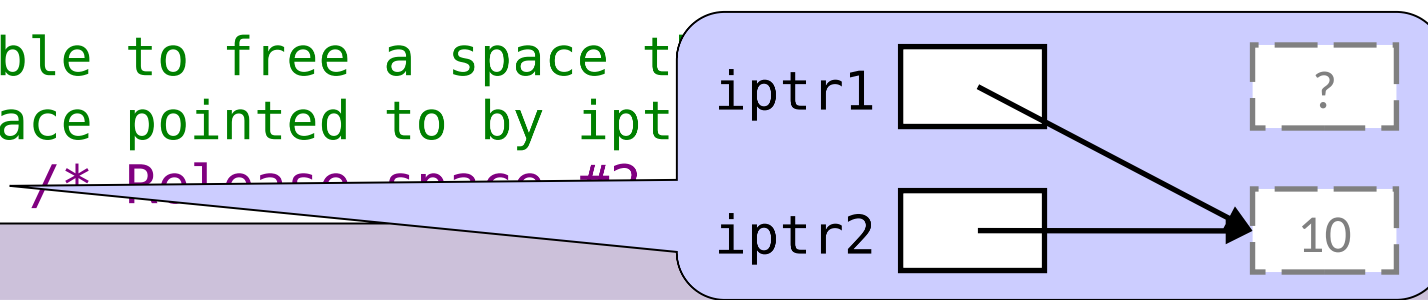
```c
int *iptr1, *iptr2;

/* Allocate a space (#1) to store an integer.
   Initial value is undefined. */
iptr1 = (int *)malloc(sizeof(int));

/* Allocate another space (#2) to store an integer
   and initialize its value to 10. */
iptr2 = (int *)malloc(sizeof(i
*iptr2 = 10;
```




```c
/* Free the space pointed to by iptr
free(iptr1);   /* Release space #1 */

/* Make iptr1 and iptr2 point to
   the same location (space
iptr1 = iptr2;
```




```c
/* It is possible to free a space t
   Free the space pointed to by ipt
free(iptr1);   /* Release space #2
```



Common Mistakes in Using `free()`

```c
int x;
int *ptr;
ptr = &x;
free(ptr);
```

(Runtime error) Cannot free a piece of memory that is not allocated using `malloc()`.

```c
int *ptr;

free(ptr);
```

Runtime error if `ptr` is not NULL.

Cannot free a space that is not allocated using `malloc()`.

```c
int *ptr = NULL;

free(ptr);
```

`free()` won't do anything if the pointer is NULL.

Common Mistakes in Using `free()`

```
int *ptr;

ptr = (int *)malloc(sizeof(int));
free(ptr);
…
*ptr = 10;
```

Dangerous to use freed space because the freed space could have been re-allocated to someone elsewhere.

```
int *ptr;
ptr = (int *)malloc(sizeof(int));
free(ptr);
free(ptr);
```

(Runtime error) Cannot free a freed space.

Dynamic VS Automatic

- Dynamic memory:
  - **manual** allocation and de-allocation
  - memory size determined at **run-time**
  - after de-allocation, the block of memory is made available for further dynamic allocation
  - IMPORTANT: manually `free` memory blocks after use! Why?
- Automatic variable:
  - **automatic** creation and destruction
  - memory size determined at **compile-time**

Dynamic Memory for Structures

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    double x, y;                    /* Define a structure type */
} Coordinates;

int main(void) {
    Coordinates point1 = {3.4, -5.9}, *ptr;
                        /* Declare a structure point1 and a pointer ptr */
    ptr = &point1;
    ptr->x = 3.458;                 /* Modify point1 via ptr */
    ptr->y = -5.967;

    ptr = (Coordinates *)malloc( sizeof(Coordinates) );
    if ( ptr == NULL )
        return 0;    /* Create ANOTHER structure dynamically and manipulate it */

    ptr->x = 47.57;
    ptr->y = 23.45;

    free(ptr);                      /* Free it after use */
    return 0;
}
```

Summary

- Dynamic memory allocation
- Dynamic memory de-allocation
- Dynamic memory VS Automatic variable
- Dynamic memory for creating structure
- Optional/ advanced topic:
  - Self-referential structure
  - Dynamic memory for creating 1D and 2D arrays

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel

- Chapter 7 C Pointers
  - Section 7.7: `sizeof` Operator
  - Sections 7.8 – 7.9: Pointer Arithmetic & Arrays

- Chapter 12 C Data Structures
  - Section 12.2: Self-Referential Structures
  - Section 12.3: Dynamic Memory Allocation

Self-Referential Structures

(Optional)

- A self-referential structure is a structure that contains a pointer member that points to a structure of the same structure type.

- Example:

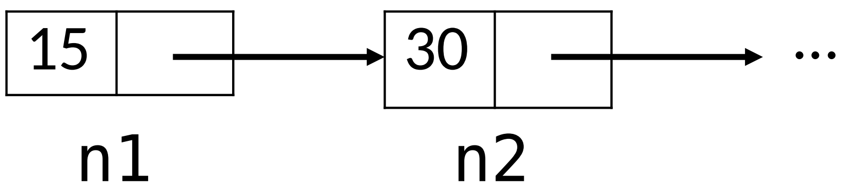
```
struct node {
    int data;
    struct node *nextptr;
};
```

Self-Referential Structures

(Optional)

```
struct node {
    int data;
    struct node *nextptr;
};
```

- The member `nextptr` can be used as a link to "tie" a `struct node` structure to another structure of the same type.



Self-Referential Structures

(Optional)

```
#include <stdio.h>

struct node {
   int data;
   struct node *nextptr;
};

int main(void) {
   struct node n1, n2;

   n1.data = 15;
   n1.nextptr = &n2;

   n2.data = 30;
   n2.nextptr = NULL;

   return 0;
}
```

- `NULL` is used when the pointer points to nothing.
- Self-referential structures can form useful data structures like lists, queues, ...

Self-Referential Structures

(Optional)

- Dynamic memory allocation can be used such that a data structure contains varying number of nodes as required.

- A node can be created dynamically when needed.

```
struct node *ptr;
……
ptr = (struct node *)malloc(sizeof(struct node));
```

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *nextptr;
};

int main(void) {
    struct node n1, *ptr;

    n1.data = 15;
    n1.nextptr = NULL;

    ptr = (struct node *)malloc(sizeof(struct node));
    ptr->data = 30;
    ptr->nextptr = NULL;

    n1.nextptr = ptr;
    ……
    free(ptr);
    return 0;
}
```

Dynamically Allocate a 1-D Array

(Optional)

Program dyn_4.c

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int i, num, *ptr, *baseptr;

   printf("How many integers? ");
   scanf("%d", &num);

   baseptr = (int *)malloc( sizeof(int) * num );
   if ( baseptr == NULL )
      return 0;                    /* Allocate memory for a few integers */

   printf("Enter %d integers: ", num);
   ptr = baseptr;
   for (i = 0; i < num; i++) {
      scanf("%d", ptr);            /* Read integers into the memory block */
      ptr++;                       /* pointer arithmetic: memory address calculation */
   }
   ......
```

Dynamically Allocate a 1-D Array

(Optional)

Program dyn_4.c (continued)

```
    ......

    printf("The numbers are: ");
    ptr = baseptr;
    for (i = 0; i < num; i++) {
        printf("%d ", *ptr);          /* Print the integers in the memory block */
        ptr++;
    }
    printf("\n");

    free(baseptr);          /* Release the memory block pointed by the Base Pointer */
    return 0;
}
```

```
How many integers? 10↵
Enter 10 integers: 9 7 45 3 222 11 66 99 77 -199↵
The numbers are: 9 7 45 3 222 11 66 99 77 -199
```

Dynamically Allocate a 1-D Array

(Optional)

- `baseptr` (the **Base Pointer**) always keeps the Base Address of the memory block allocated by `malloc()`.

- We must `free` the WHOLE memory block at once, with this Base Address.

- `ptr` (the **Working Pointer**) “moves” in the memory block, pointing to different locations storing integers.

- Alternative technique:
  use `baseptr[i]` or `*(baseptr + i)`
  instead of `ptr`.

(Optional)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int i, num, *baseptr;

   printf("How many integers? ");
   scanf("%d", &num);

   baseptr = (int *)malloc( sizeof(int) * num );
   if ( baseptr == NULL )
      return 0;

   printf("Enter %d integers: ", num);
   for (i = 0; i < num; i++)
      scanf("%d", &baseptr[i]);

   printf("The numbers are: ");
   for (i = 0; i < num; i++)
      printf("%d ", baseptr[i]);
   printf("\n");

   free(baseptr);
   return 0;
}
```

Alternative technique: treating `baseptr` as a "dynamic array"

Or: `baseptr + i`

Or: `*(baseptr + i)`

(Optional)

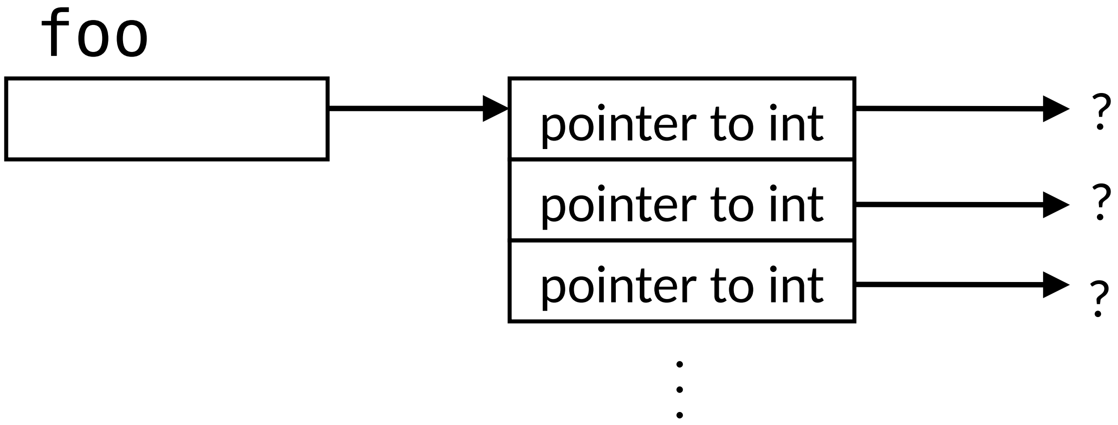
```
int *array, *tmp, i, n;

n = …;      /*  Suppose n is determined here  */

array = (int *)malloc(n * sizeof(int));

…   /*  Elements of array have been assigned some values here…  */

/*  Suppose now we want to enlarge array to hold 2n integers, and we want
    to retain the existing data. Solution:
    (1) Allocate a larger array,
    (2) copy the data from array to the newly allocated array,
    (3) free up the space occupied by array, and
    (4) makes array points to the newly allocated array  */

  tmp = (int *)malloc(2 * n * sizeof(int));   /* (1) */
  for (i = 0; i < n; i++)                      /* (2) */
     tmp[i] = array[i];
  free(array);                                 /* (3) */
  array = tmp;                                 /* (4) */
```

Example: Enlarging a dynamically allocated array.

Dynamically Allocate a 2-D Array

(Optional)

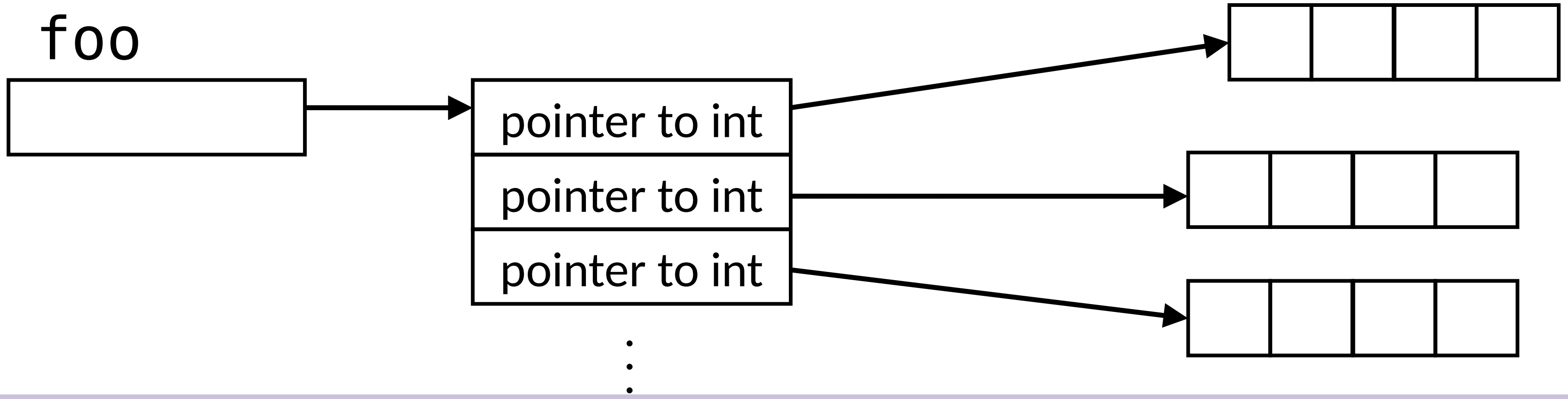
```
int **foo;
int ROW = 10;

/* foo is considered as an array of pointers */
foo = (int **)malloc(ROW * sizeof(int *));
/* Each element is a pointer to int */
```



Dynamically Allocate a 2-D Array

(Optional)

```
int **foo;
int i, ROW = 10, COL = 4;

foo = (int **)malloc(ROW * sizeof(int *));
/* Each element is a pointer to int */

/* Allocate a 1-D array for each row */
for (i = 0; i < ROW; i++)
    foo[i] = (int *)malloc(COL * sizeof(int));
/* Now foo can be used as a ROW × COL 2-D array of int */
```



Allocating/Deallocating 2-D Arrays

(Optional)

```
int **foo;
int i, j, ROW = 10, COL = 4;

/* Allocate space for an array of pointers  */
foo = (int **)malloc(ROW * sizeof(int *));

/* Allocate space for each row (which is a 1-D array) */
for (int i = 0; i < ROW; i++) {
   foo[i] = (int *)malloc(COL * sizeof(int));
   for (j = 0; j < COL; j++)
      foo[i][j] = i + j;   /* Can be used as a 2D array */
}
…
/* Free the space of each row first */
for (int i = 0; i < ROW; i++)
   free(foo[i]);

free(foo);   /* Finally, free the array of pointers */
```

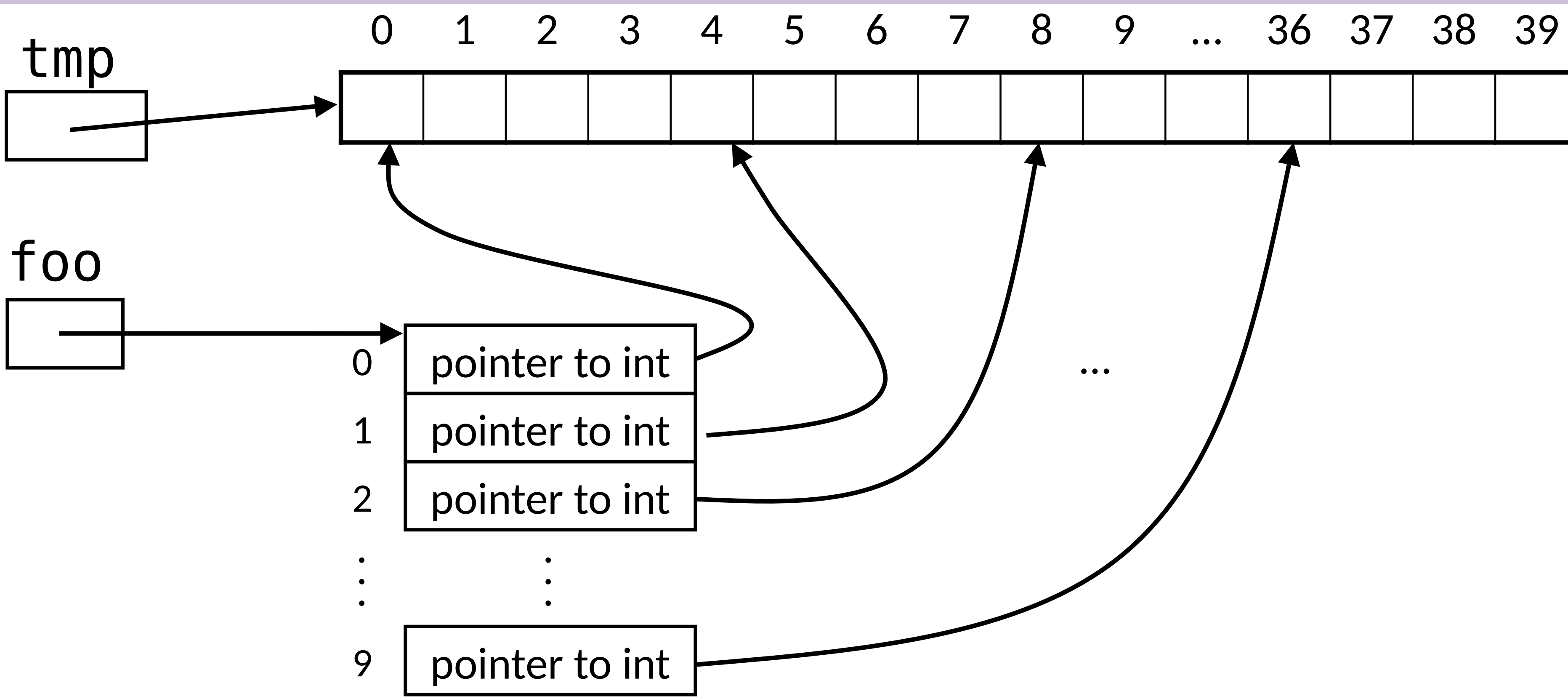
Allocating/Deallocating 2-D Arrays

(Optional)

```
int **foo;
int *tmp;
int i, j, ROW = 10, COL = 4;

/* Allocate space for an array of pointers  */
foo = (int **)malloc(ROW * sizeof(int *));

/* Alternative approach
   Allocate a 1-D array big enough for all elements in the
   2-D array and then share the space among all rows */
tmp = (int *)malloc(ROW * COL * sizeof(int));
for (i = 0; i < ROW; i++)
    foo[i] = &tmp[COL * i];


…
free(tmp);
free(foo);
```

Allocating/Deallocating 2-D Arrays

(Optional)



Allocate a 1-D array (`tmp`) big enough for <u>all</u> elements in the 2-D array and then share the space among all rows