Comparing pointers

- pointers can be compared for equality or inequality
  - two pointers are equal if
    - they both point to the same object, or
    - they both point one past the end of the same array, or
    - they both have the value **`NULL`**

- if the pointer types are different, then the compiler will output a warning
  - but there is nothing stopping you from comparing **`void *`** pointers

```c
#include<stdio.h>
// compare_ptr.c
int main(void) {
    int i = 0;
    int *p = &i;
    int *pi = &i;

    if (p == pi) {
        puts("p points to same object as pi");
    }

    return 0;
}
```

```c
#include<stdio.h>
// compare_ptr_warning.c
int main(void) {
    int i = 0;
    double d = 0.0;

    int *pi = &i;
    double *pd = &d;

    if (pi == pd) {    // compiler warning
        puts("pi points to same object as pd");
    }
    else {
        puts("pi points to a different object than pd");
    }

    return 0;
}
```

The too-far pointer

- C guarantees that a pointer can point to the element one position past the end of an array
  - called the too-far pointer
  - however, dereferencing this pointer results in undefined behavior
- needed because older C code frequently incremented a pointer until it reached the too-far pointer

```c
#include<stdio.h>
// compare_ptr2.c
int main(void) {
    int arr[] = {1, 2, 3};
    int *toofar = &arr[0] + 3;    // one past the end of the array
    int *p = &arr[0];
    int *q = &arr[0];
    while (p != toofar) {
        // do something with p and q here, then advance pointers
        p++;
        q++;
    }
    if (p == q) {
        puts("p and q are equal");
    }
    return 0;
}
```

Comparing pointers

- a null pointer is a pointer that points to no object
  - dereferencing such a pointer results in undefined behavior
- a null pointer can be created using the null pointer constant `NULL` or integer zero

```c
#include<stdio.h>
// compare_nullptr.c
int main(void) {
    int *p = NULL;
    int *q = NULL;

    if (p == q) {
        puts("p and q are equal");
    }
    else {
        puts("p and q are not equal");
    }

    return 0;
}
```

Testing for null

- there are three valid ways to test if a pointer **`p`** is null:

```
if (!p)

if (p == NULL)

if (p == 0)
```

```c
#include<stdio.h>
// test_for_null.c
int main(void) {
    int *p = NULL;    // a null pointer

    if (!p) {
        puts("p is a null pointer");
    }
    if (p == NULL) {
        puts("p is a null pointer");
    }
    if (p == 0) {
        puts("p is a null pointer");
    }

    // avoid doing the following
    int null = 0;
    if (p == null) {
        puts("p is a null pointer");
    }

    return 0;
}
```

Comparing `void *` pointers

- `void *` pointers can be tested for equality

```c
#include<stdio.h>
// compare_voidptr.c
int main(void) {
    int i = 0;
    double d = 0.0;

    void *p = &d;
    void *pi = &i;
    void *pd = &d;

    if (pi == pd) {    // no compiler warning
        puts("pi points to same object as pd");
    }
    if (p == pd) {
        puts("p points to same object as pd");
    }

    return 0;
}
```

Comparing pointers

- the operators **<**, **<=**, **>**, and **>=** can also be used to compare pointers
  - however, both pointers should be pointers into the same array
- if two pointers point to different elements of the same array, the one pointing at the element with the larger index compares greater
- if one pointer points to the element of an array and the other pointer points one past the end of the same array, the one-past-the-end pointer compares greater

```c
#include<stdio.h>
// compare_ptr3.c
int main(void) {
    int arr[] = {1, 2, 3};

    int *p = &arr[0];    // points to p[0]
    int *q = p + 1;      // points to p[1]
    int *r = p + 3;      // points to too-far

    if (r > q) {
        puts("r is greater than q");
    }
    if (q > p) {
        puts("q is greater than p");
    }

    return 0;
}
```

Variable length arrays

- C99 formally introduced variable length arrays
- "variable length" does not mean that the size of the array can change during the lifetime of the array
  - the size of an array is a constant during the lifetime of the array
- variable length means the that the length of the array is defined by a variable instead of a constant

```c
#include<stdio.h>
// vla1.c
void print(size_t n, int arr[n]) {
    if (n == 0) {
        puts("[]");
    }
    else {
        printf("[%d", arr[0]);
        for (size_t i = 1; i < n; i++) {
            printf(", %d", arr[i]);
        }
        puts("]");
    }
}
```

```c
int main(void) {
    puts("Enter an array size less than 20");
    size_t n = 0;
    int result = scanf("%lu", &n);
    if (result == 1 && n < 20) {
        int arr[n];
        for (int i = 0; i < n; i++) {
            arr[i] = i;
        }
        print(n, arr);
    }
    return 0;
}
```

Notes

- **`size_t`** is the unsigned integer type of the result of the **`sizeof`** operator
- can store the maximum size of a theoretically possible object of any type (including arrays)
- **`scanf`** reads data from standard input and tries to convert the data into the format specified by the formatting string
  - if successful, it stores the data in the arguments that follow the formatting string
  - returns the number of successfully converted arguments, or **`EOF`** if input failure occurs before the first receiving argument was assigned

Variable length arrays

- regular arrays and variable length arrays have automatic storage
  - their lifetime is the lifetime of the block that they are defined in
- this means that you cannot safely return a regular array or variable length array from a function
  - because the array no longer exists after the function returns

```c
#include<stdio.h>
// array_lifetime.c
void print(size_t n, int arr[n]) {  // same as before
}

int *make_array(size_t n) {
    int arr[n];                     // uhoh, automatic storage
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
    return arr;
}

int main(void) {
    puts("Enter an array size less than 20");
    size_t n = 0;
    int result = scanf("%lu", &n);
    if (result == 1 && n < 20) {
        int *arr = make_array(n);
        print(n, arr);
    }
    return 0;
}
```

Dynamically allocated memory

- data structures such as stacks, queues, lists, sets, and maps are usually implemented so that they can grow and shrink as elements are added and removed
- the memory for such data structures should not have automatic storage duration
  - why not?
- the memory for such data structures should not have static storage duration
  - why not?

Dynamically allocated memory

- dynamically allocated memory has *allocated storage duration*
  - the lifetime of an allocated object begins when the allocation function returns and ends when the deallocation function is called
- dynamically allocated memory is managed by a memory manager, which is not part of the C language specification
  - memory manager implementations depend on the operating system and system architecture
    - some applications even implement their own memory managers
  - responsible for allocating memory, and managing deallocated memory

The heap

- common implementations of the allocator use a heap, which is one or more subdividable blocks of memory under control of the memory manager
- the allocator services requests for memory by allocating a block of unused memory, marking the block as used, and returning a pointer to the memory block
  - the requester is then responsible for managing the memory
  - it is the requesters responsibility to deallocate, or free, the memory when it is no longer needed

When to use allocated memory?

- dynamically allocated memory is used when the exact memory requirements are not known before runtime
- dynamically allocated memory is less efficient than statically allocated memory because the memory manager needs to allocate memory from the heap
- memory leaks occur when the requester neglects to return unused memory back to the memory manager

Dynamic memory manangement

- **`<stdlib.h>`** declares three functions for requesting dynamically allocated memory
  - four functions for C11
  1. **`malloc`**
  2. **`calloc`**
  3. **`realloc`**
- one function for releasing allocated memory
  1. **`free`**

`malloc`

- attempts to allocate an *uninitialized* block of memory of a specified size
  - remember that sizes are measured in terms of the size of **`char`** in C
- returns a pointer to the allocated memory, or a null pointer if the request could not be fulfilled

```c
#include<stdio.h>
#include<stdlib.h>
// array_malloc.c
void print(size_t n, int arr[n]) {    // same as previous
}

int *make_array(size_t n) {
    int *arr = malloc(n * sizeof(int));
    return arr;
}

int main(void) {
    puts("Enter an array size less than 20");
    size_t n = 0;
    int result = scanf("%lu", &n);
    if (result == 1 && n < 20) {
        int *arr = make_array(n);
        print(n, arr);
        free(arr);
    }
    return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
// array_malloc.c
void print(size_t n, int arr[n]) {    // same as previous
}

int *make_array(size_t n) {
    int *arr = malloc(sizeof(int[n]));   // or specify the size of an array
    return arr;
}

int main(void) {
    puts("Enter an array size less than 20");
    size_t n = 0;
    int result = scanf("%lu", &n);
    if (result == 1 && n < 20) {
        int *arr = make_array(n);
        print(n, arr);
        free(arr);
    }
    return 0;
}
```

`calloc`

- attempts to allocate an block of memory for *n* objects each having a specified size, and then initializes all bytes in the allocated memory to zero
  - remember that sizes are measured in terms of the size of **`char`** in C
- returns a pointer to the allocated memory, or a null pointer if the request could not be fulfilled

```c
#include<stdio.h>
#include<stdlib.h>
// array_malloc.c
void print(size_t n, int arr[n]) {    // same as previous
}

int *make_array(size_t n) {
    int *arr = calloc(n, sizeof(int));
    return arr;
}

int main(void) {
    puts("Enter an array size less than 20");
    size_t n = 0;
    int result = scanf("%lu", &n);
    if (result == 1 && n < 20) {
        int *arr = make_array(n);
        print(n, arr);
        free(arr);
    }
    return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
// array_calloc.c
void print(size_t n, int arr[n]) {    // same as previous
}

int *make_array(size_t n) {
    int *arr =  calloc(1, sizeof(int[n]));   // or specify the size of an array
    return arr;
}

int main(void) {
    puts("Enter an array size less than 20");
    size_t n = 0;
    int result = scanf("%lu", &n);
    if (result == 1 && n < 20) {
        int *arr = make_array(n);
        print(n, arr);
        free(arr);
    }
    return 0;
}
```