`realloc`

- attempts to reallocate an block of memory that was previously allocated by **`malloc`**, **`calloc`**, or **`realloc`**
  - the memory must not have been freed
  - **`void *realloc(void *ptr, size_t new_size)`**
    - **`ptr`** : pointer to memory are to be reallocated
    - **`new_size`** : non-zero, new size of array in bytes (undefined behavior if zero)
- on success, returns a pointer to the newly allocated memory
- on failure, returns a null pointer
  - original pointer remains valid and should still be deallocated using **`free`** when no longer needed

realloc

- if reallocation succeeds, then the contents of the original array that fit into the reallocated memory are preserved
  - any excess memory is not initialized

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

int *make_array(size_t n) {
    int *arr = malloc(sizeof(int[n]));
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
    return arr;
}
```

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

typedef

`typedef`

- the **`typedef`** declaration provides a way to declare an identifier as a type alias, to be used to replace a possibly complex type name
  - does not create a new type, simply says that the alias can be used in place of the actual type
- syntax:

  `typedef` *`type alias`*

  where *`type`* is an existing type and *`alias`* is the new alias for the existing type

Some types in the standard library are actually aliases created using **`typedef`**:

```
typedef long unsigned int size_t;  // size of an object

typedef long int ptrdiff_t;        // pointer difference
```

`typedef`

- `typedef` is usually used to simplify writing complicated types and to provide incomplete types

```c
// int_ptr_t is a pointer to an int
typedef int *int_ptr_t;

// f_ptr_t is a pointer to a function that returns
// an int and has two int parameters
typedef int (*f_ptr_t)(int, int);

// list is an alias for struct list
typedef struct list list;
```

```c
#include<stdio.h>
#include<stdlib.h>
// array_malloc_typedef.c
typedef int *int_ptr_t;

void print(size_t n, int arr[n]) {  // same as previous
}

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

```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_ptr_t arr = make_array(n);
        print(n, arr);
        free(arr);
    }
    return 0;
}
```

Structures

Structures

- a structure, or struct, consists of one or members whose storage is allocated in an ordered sequence
  - i.e., a struct is a group of variables in one block of memory having a single name
- sort of like a Java class having no methods
- syntax

```
struct tag_name {
    type member1;
    type member2;
    // and so on
};
```

A two-dimensional point struct:

```c
#include<stdio.h>
// struct_point2.c
struct point2 {
    double x;
    double y;
};

int main(void) {
    struct point2 p;
    printf("%f, %f\n", p.x, p.y);
}
```

Type name of a struct

- the tag name of a struct is not a type
- instead, tags exist a different namespace than identifiers such as variables, typedef names, and function names
  - this means that you can have a tag and a variable with the same name
- you can think of

  `struct` *`tagname`*

  as being the type of a struct
  - usually easier to use a **`typedef`**

A two-dimensional point struct:

```c
#include<stdio.h>
// struct_point2.c
struct point2 {
    double x;
    double y;
};

int main(void) {
    struct point2 p;
    printf("%f, %f\n", p.x, p.y);
}
```

Same example using a **`typedef`**:

```
#include<stdio.h>
// struct_point2.c
struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

int main(void) {
    point2 p;
    printf("%f, %f\n", p.x, p.y);
}
```

Initializing members

- struct members can be initialized using a syntax similar to array initialization
  - the values for the members can be given as a comma separated list inside of braces
    - list of initializers cannot be empty
  - the member variable names are optional
    - must preceded by a `.` if given
      - order does not need to match the order that the members are listed if the member variables are given
- members not explicitly initialized are zero-initialized

```c
#include<stdio.h>
// struct_point2_2.c
struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

int main(void) {
    point2 p = { 2.0, 3.0 };
    printf("p: %f, %f\n", p.x, p.y);

    point2 q = { .x = 1.0, .y = 1.5 };
    printf("q: %f, %f\n", q.x, q.y);

    point2 r = { .y = -3.1 };
    printf("r: %f, %f\n", r.x, r.y);
}
```

Accessing members

- if you have a struct object, you can access a member using the `.` operator
- if you have a pointer to a struct object, you can access a member using the `->` operator
  - dereferences the pointer to struct and then accesses the member

```c
#include<stdio.h>
// struct_point2_3.c
struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

int main(void) {
    point2 p = { 2.0, 3.0 };
    printf("p: %f, %f\n", p.x, p.y);

    point2 *ptr = &p;
    ptr->x = 200.0;
    ptr->y = 300.0;
    printf("p: %f, %f\n", p.x, p.y);
}
```

Assignment

- a struct may be assigned to another struct
- result is a member-wise copy
  - if the struct contains a pointer, then you have two struct objects with a pointer that points to the same object

```c
#include<stdio.h>
// struct_point2_4.c
struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

int main(void) {
    point2 p = { 2.0, 3.0 };
    printf("p: %f, %f\n", p.x, p.y);

    point2 q;
    printf("q: %f, %f\n", q.x, q.y);

    q = p;
    printf("q: %f, %f\n", q.x, q.y);

    q.x = 200.0;
    q.y = 300.0;
    printf("p: %f, %f\n", p.x, p.y);
    printf("q: %f, %f\n", q.x, q.y);
}
```

Dynamic allocation of struct

- memory for a struct may be dynamically allocated
  - simply use **`sizeof`** to get the struct size

```c
#include<stdio.h>
#include<stdlib.h>
// struct_point2_5.c
struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

int main(void) {
    point2 *p = malloc(sizeof(point2));
    p->x = 100.0;
    p->y = 200.0;
    printf("p: %f, %f\n", p->x, p->y);
}
```

Comparing structs for equality

- C has no built-in mechanism for comparing structs for equality
  - `==` and `!=` are not defined for structs
- if you need to do this, then you have to decide what equality means for your struct type and then compare the relevant members
  - if you need to do this more than once, then you should write a function

```c
#include<stdio.h>
#include<stdlib.h>
// struct_point2_6.c
struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

int main(void) {
    point2 *p = malloc(sizeof(point2));
    p->x = 100.0;
    p->y = 200.0;

    point2 q = { p->x, p->y };
    if ((*p).x == q.x && (*p).y == q.y) {
        puts("*p and q objects have the same coordinates");
    }
}
```

Organization of a C program

Header files

- when creating a new type such as **`point2`**, it is often the case that you create functions to accompany the type
  - in Java, you would bundle the fields and methods into a class
- in C, you can place the type declaration and function declarations in a file called a *header* file
  - has extension **`.h`**
  - any C source code file that uses your type includes the header file for your type
- the definition of the functions are placed in a separate C source code file

`point2` functions

- `point2_move(point2 *p, double dx, double dy)`
  - moves a point
- `point2_mult(point2 *p, double s)`
  - multiplies coordinates by s
- `point2_equals(const point2 p, const point2 q)`
  - test if two points have the same coordinates
- `point2_to_string(const point2 p)`
  - returns a string representation of a point

Include guard

- every header file should have an include guard
  - prevents the header file from being included more than once in a compilation unit
- traditionally implemented using preprocessor directives
  - https://en.cppreference.com/w/c/preprocessor

```
#ifndef POINT2_H
#define POINT2_H



#endif // POINT2_H
```

if the identifier **POINT2_H** is not defined, then continue processing the header file; otherwise jump past the #endif

define the identifier **POINT2_H**

ends **ifndef**

```c
#ifndef POINT2_H
#define POINT2_H
#include <stdbool.h>

struct point2 {
    double x;
    double y;
};
typedef struct point2 point2;

void point2_move(point2 *p, double dx, double dy);

void point2_mult(point2 *p, double s);

bool point2_equals(const point2 p, const point2 q);

char *point2_to_string(const point2 p);

#endif // POINT2_H
```

Source code file

- a separate C source code file contains the definition of the functions

```c
#include <stdio.h>
#include <stdlib.h>
#include "point2.h"
```

Header files that are not part of the C standard library are included using `""`

```c
#include <stdio.h>
#include <stdlib.h>
#include "point2.h"

void point2_move(point2 *p, double dx, double dy) {
    p->x += dx;
    p->y += dy;
}

void point2_mult(point2 *p, double s) {
    p->x *= s;
    p->y *= s;
}

bool point2_equals(const point2 p, const point2 q) {
    // does not handle NaN coordinates correctly
    return p.x == q.x && p.y == q.y;
}
```

```c
char *point2_to_string(const point2 p) {
    // allocate a buffer large enough for returned string
    // ~15 chars for a double using scientific notation
    // 2 chars for leading ( and trailing )
    // 2 chars for , separator
    const int BUF_SZ = 15 * 2 + 4;
    char *buf = malloc(BUF_SZ);
    if (!buf) {
        return NULL;
    }
    int n = sprintf(buf, "(%g, %g)", p.x, p.y);
    if (n < 0 || n >= BUF_SZ) {
        free(buf);
        return NULL;
    }
    return buf;
}
```