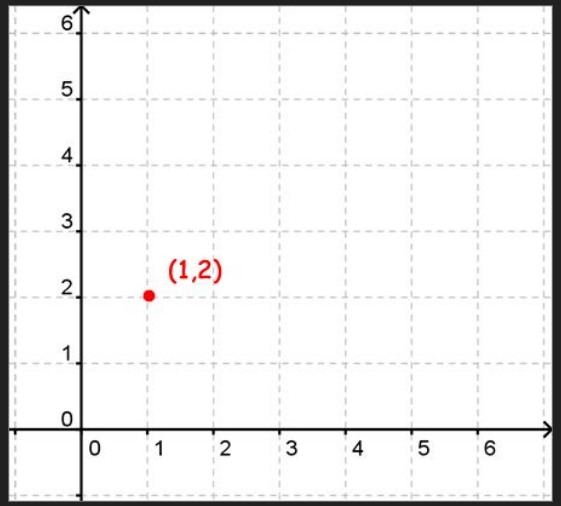
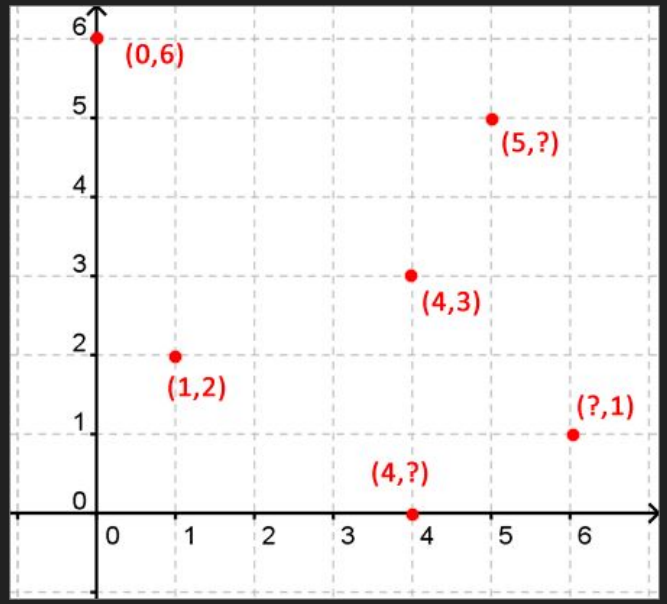
Structs

- represent some real world **structure**
- store in a contiguous block of memory; **in order**
- can use multiple data types!



Structs




```
int point[2] = {1,2};
int p2[2] = {0, 6};
```

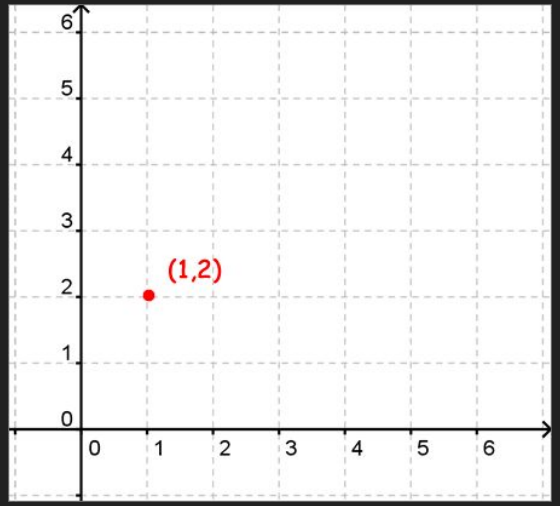
```
point_t pnt = {1,2};
int p2[2] = {0, 6};

pnt.x pnt.y
```

```c
struct point {
    int x, y;
};

int main() {
    struct point pnt1 = {1, 2};
    struct point pnt2 = {2, 3};

    printf("(%d, %d)", pnt1.x, pnt1.y);
    // (1, 2)
    printf("(%d, %d)", pnt2.x, pnt2.y);
    // (2, 3)
}
```



```c
typedef struct {
    int x, y;
} point_t;

int main() {
    point_t pnt1 = {1, 2};
    point_t pnt2 = {2, 3};

    printf("(%d, %d)", pnt1.x, pnt1.y);
    // (1, 2)
    printf("(%d, %d)", pnt2.x, pnt2.y);
    // (2, 3)
}
// DRY : Don't Repeat Yourself -- I can write a function!
```

```c
typedef struct {
    int x, y;
} point_t;

void point_print(point_t p) { printf("(%d, %d)", p.x, p.y); }

int main() {
    point_t pnt1 = {1, 2};
    point_t pnt2 = {2, 3};

    point_print(pnt1);
    // (1, 2)
    point_print(pnt2);
    // (2, 3)
}
```

```c
typedef struct { int x, y; } point_t;
void point_print(point_t p) { printf("(%d, %d)", p.x, p.y); }

point_t point_add(point_t p1, point_t p2) {
    point_t newp;    //point_t newp = p1;  // shallow copy
    newp.x = p1.x + p2.x;
    newp.y = p1.y + p2.y;
    return newp;
}

int main() {
    point_t pnt1 = {1, 2}; point_t pnt2 = {2, 3};
    point_t res = point_add(pnt1, pnt2);
    point_print(res);
    // (3, 5)
}
```

```c
typedef struct { int x, y; } point_t;
void point_print(point_t p) { printf("(%d, %d)", p.x, p.y); }

point_t point_add(point_t p1, point_t p2) {
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}

int main() {
    point_t pnt1 = {1, 2}; point_t pnt2 = {2, 3};

    point_print(point_add(pnt1, pnt2));
    // (3, 5)
    point_print(pnt1); // (1, 2)
}
```

```c
typedef struct { int x, y; } point_t;
void point_print(point_t p) { printf("(%d, %d)", p.x, p.y); }

point_t point_add(point_t p1, point_t p2) {
    p1.x += p2.x;        // ^ also being shallow copied...
    p1.y += p2.y;
    return p1;
}

int main() {
    point_t pnt1 = {1, 2}; point_t pnt2 = {2, 3};
    point_t result = point_add(pnt1, pnt2);  // shallow copy!
    point_print(result);
    // (3, 5)
}
```

```c
typedef struct { int x, y; } point_t;
void point_print(point_t p) { printf("(%d, %d)", p.x, p.y); }

point_t point_add(point_t *p1, point_t *p2) {
    point_t newp = *p1;
    newp.x = (*p1).x + p2->x; // (*p1).x == p1->x
    newp.y = p1->y + p2->y;
    return newp;
}

int main() {
    point_t pnt1 = {1, 2}; point_t pnt2 = {2, 3};
    point_t result = point_add(&pnt1, &pnt2);
    point_print(result);
    // (3, 5)
}
```

if we take &obj, we can access obj.x with:
// access member variable of a pointer struct
obj->x == (*obj).x

Beware of operator precedence...
obj->x != *obj.x == *(obj.x)

```c
typedef struct { int x, y; } point_t;

// Shallow copies the point (2 ints copied, 8 bytes)
void point_print(point_t p) { printf("(%d, %d)", p.x, p.y); }

// Only a pointer copied!  (8 bytes or 4 bytes)
void point_printp(point_t *p) {
    printf("(%d, %d)", p->x, p->y);
}
int main() {
    point_t pnt = {1, 2};
    point_print(result);        // (shallow) Copies the struct
    point_printp(&result);      // passes the struct by pointer
    // Both have the same result
}
```

Structs are not arrays.

```
typedef struct {
   int x, y;
} point_t;

!=

typedef int point_t[2];
```

Structs are not arrays.

```
typedef struct {
   int x, y;
} point_t;
```

!=

```
typedef int point_t[2];
```

both look similar in memory!

```
// both can be declared like this
point_t p = {1, 2};

// struct
printf("(%d, %d)", p.x, p.y);

// array
printf("(%d, %d)", p[0], p[1]);

// C99 Designated Initialiser
point_t p = {.y = 2, .x = 1};
```

```
typedef struct {int x, y;} point_t;

// Find the point with the maximum y value.
point_t highest_point(point_t points[], int n) {
    assert(n > 0);

    point_t highest = points[0];
    for (int i = 1; i < n; i++) {
        if (points[i].y > highest.y) {
           highest = points[i];   // shallow copy structs
        }
    }
    return highest;
}
```

points[i].y
*(points + i).y
(points + i)->y

```
typedef struct {int x, y;} point_t;


// Find the point with the maximum y value.
point_t highest_point(point_t points[], int n) {
    assert(n > 0);

    point_t highest = points[0];
    for (int i = 1; i < n; i++) {
        if ((points+i)->y > highest.y) {   // if you want!
           highest = points[i];
        }
    }
    return highest;
}
```

contrived example...

```
typedef struct {int x, y;} point_t;
typedef struct {int idx; point_t max;} point_idx_t;

// Find the point with the maximum y value, along with its index
point_idx_t highest_point(point_t points[], int n) {
    assert(n > 0);

    int highest_idx = 0;
    for (int i = 1; i < n; i++) {
        if (points[i].y > points[highest_idx].y) {
            highest_idx = i;
        }
    }
    point_idx_t ret = {.pt=points[highest_idx], .idx=highest_idx};
    return ret;
}
```

Hierarchical struct

Struct representing a user's bank account

```
#define NAMELEN 20

typedef int cents_t;
typedef char name_t[NAMELEN+1];

typedef struct {
    int id;
    name_t name;
    int opened_year, opened_month, opened_date; // yy/mm/dd
    int closed_year, closed_month, closed_date;
    cents_t balance;    // STORING MONEY IN DOUBLE IS A BAD IDEA
} account_t;

DRY!  Solution?
```

Hierarchical struct

Struct representing a user's bank account

```
#define NAMELEN 20

typedef int cents_t;
typedef char name_t[NAMELEN+1];

typedef struct {
    int dd, mm, yyyy;
} date_t;

typedef struct {
    int id;
    char name[21];
    date_t opened, closed;
    double balance;
} account_t;
```

Hierarchical struct

Struct representing a user's bank account

```
typedef struct {
    int dd, mm, yyyy;
} date_t;

typedef struct {
    int id;
    char name[21];
    date_t opened, closed;
    double balance;
} account_t;
```

```
void print_date(date_t date) {
    printf("%d/%d/%d", date.dd, date.mm, date.yyyy);
}

void print_date(date_t *date) {
    printf("%d/%d/%d", date->dd, date->mm, date->yyyy);
}

// then
account_t account = {12, "Liam", {11,12,2020},
                         {12,12,2020}, 0.01};

print_date(account.opened); // 11/12/2020

print_date(account.closed); // 12/12/2020
```

Recursive struct definition

Struct representing a family tree.

```
typedef char name_t[NAME_LEN + 1];

// family tree

typedef struct {
    name_t name;
    person_t *mother;
    person_t *father;
    person_t *spouse;
} person_t;

// this won't work.  person_t is used before it is defined.
```

Recursive struct definition

Struct representing a family tree.

```
typedef char name_t[NAME_LEN + 1];

// family tree
typedef struct person person_t;

struct person {
    name_t name;
    person_t *mother;
    person_t *father;
    person_t *spouse;
};

struct person liam;   // person_t liam;
```

This will come in handy for Linked Lists and Binary Search Trees.

Recursive struct definition

Struct representing a family tree.

```
typedef char name_t[NAME_LEN + 1];

// family tree
typedef struct person person_t;

struct person {
    name_t name;
    person_t *mother;
    person_t *father;
    person_t *spouse;
};

struct person liam;   // person_t liam;
```

```
// constructor
person_t create_person(name_t name) {
    person_t p = {name, NULL, NULL, NULL};
    strcpy(p.name, name);
    return p;
}

person_t liam = create_person("Liam");
person_t dad = create_person("Dad");
person_t mum = create_person("Mum");

liam.mother = &mum;
liam.father = &dad;
```

This will come in handy for Linked Lists and Binary Search Trees.

- Use structs for...
  - Store hierarchically arranged data
  - Minimise repeated variable declarations
    - int year1, int year2, int year3
  - Simplifying function calls
  - Multiple return values from functions (less messy than pointers)

- Be careful...
  - Structs are (shallow) copied in their entirety when passed to a function; **unlike arrays**
    - For large structs, this is an expensive operation
    - We may instead copy a pointer to the structure instead
    - Every struct.parameter becomes a struct->parameter
      - or (*struct).parameter, but no-one writes it that way

void pointers

Motivation

- I want to implement an algorithm, but I don't know / care what type I'm using the algorithm with.

eg: quicksort on ints, strings, arrays of ints, arrays of strings, or structs ...

void *

- Can implement **generic** / **polymorphic** functions: work with many types.
  - eg: qsort, bsearch, ...

- Can deal with buffers of bytes.
  - eg: malloc, free, ...

void *

Pointers are the same size regardless of type

- 4 bytes -- 32 bit (1 byte is 8 bits)
- 8 bytes -- 64 bit

sizeof(int *) == sizeof(double *) == sizeof(country_t *) == sizeof(void *).

sizeof(void) = undefined "intermediate type"

so what's the point of having typed pointers?

void *

```
int x = 4;

void* p = &x;

*p = 7;              // illegal with void pointers


*((int *) p) = 7;  // correct
```

void * issues

- can't dereference void pointers
  - void *p: *p is undefined (previous slide)
- sizeof(void) is undefined
  - p[i] is undefined as p[i] => *(p + i * sizeof(void))
- void pointer arithmetic: not standard compliant (is a gcc extension)
- Need to use casts everywhere where we want to use it
  - *((int *) p) = 7;
- Nothing protecting you from casting to the wrong type

void * issues

- Nothing protecting you from casting to the wrong type

```
int x = 2;    // x is 4 bytes
int *p = &x;  // p is 8 bytes
void *p1 = p; // p1 is 8 bytes
double *p2 = p1;

sizeof(*p) = ??
sizeof(*p1) = ??
sizeof(*p2) = ??
```

void * issues

- Nothing protecting you from casting to the wrong type

```
int x = 2;    // x is 4 bytes
int *p = &x;  // p is 8 bytes
void *p1 = p; // p1 is 8 bytes
double *p2 = p1;

sizeof(*p) = sizeof(int) = 4
sizeof(*p1) = sizeof(void) = undefined!
sizeof(*p2) = sizeof(double) = 8
```