Structures

COP 3223C – Introduction to Programming with C

Fall 2025

Yancy Vance Paredes, PhD

Scenario

- Write a program that reads data about people from a file
- It has the name (max of 100 **`char`**) and the birth year of **`N`** people
- Assume that no two persons have the same birth year
- Your task is to print the name and age of the oldest person

Sample Run

people.txt

```
3
John Doe
2010
Jane Smith
2005
Robert Smith
1995
```

Sample Output

```
30 Robert Smith
```

Challenge

- Another task is to sort them by their current age (ascending)
- Write the result to a new file called **`sorted.txt`**

Sample Run

people.txt

```
3
John Doe
2010
Jane Smith
2005
Robert Smith
1995
```

sorted.txt

```
20 Jane Smith
25 John Doe
30 Robert Smith
```

Practice

Read the contents of the file

Print the information on the screen

Parallel Arrays /1

- The idea is to have an array for each information about the person

- For example, one for name then another for birth year, and so on

- We “connect” them through the **index**; thus, they are parallel

Parallel Arrays /2

2

char names [MAX_SIZE][MAX_LEN]

i	0	1	2	3	4
names[i]	"John Doe"	"Jane Smith"	"Robert Smith"	?	?

int years [MAX_SIZE]

i	0	1	2	3	4
years[i]	2010	2005	1995	?	?

Strategy

- The goal is to find the index of the min value in the **`years`** array
- Once found, that index (i.e., position) is used for the **`names`** array
- Then, we print the result
- For now, let’s write the solution in the **`main()`** function

Practice

Write a **`find_min()`** function that returns the index of the maximum element in an array with given size

```
int find_min(int *arr, int size) {



}
```

Refer to Webcourses for the C Code: People Database (Parallel Arrays)

Discussion

- For each person, we are keeping track of two information
- What if we want to add more information to track?
- We can still do parallel arrays; one array for each new information
- However, there may be another approach

User-Defined Structure Types /1

- An approach to **organizing** data

- The idea is to **logically** *group variables* (usually of different types)

- Notice that these variables are related to each other

- Essentially, you are creating a new *data type*

User-Defined Structure Types /2

- Notice how all the variables are related to a person?

- Why don't we create a new data type?

- These variables are known as **members** or **components**

fields

User-Defined Structure Types /3

Syntax:

member or components

fields

Practice

- Define a struct called **`Person_s`**

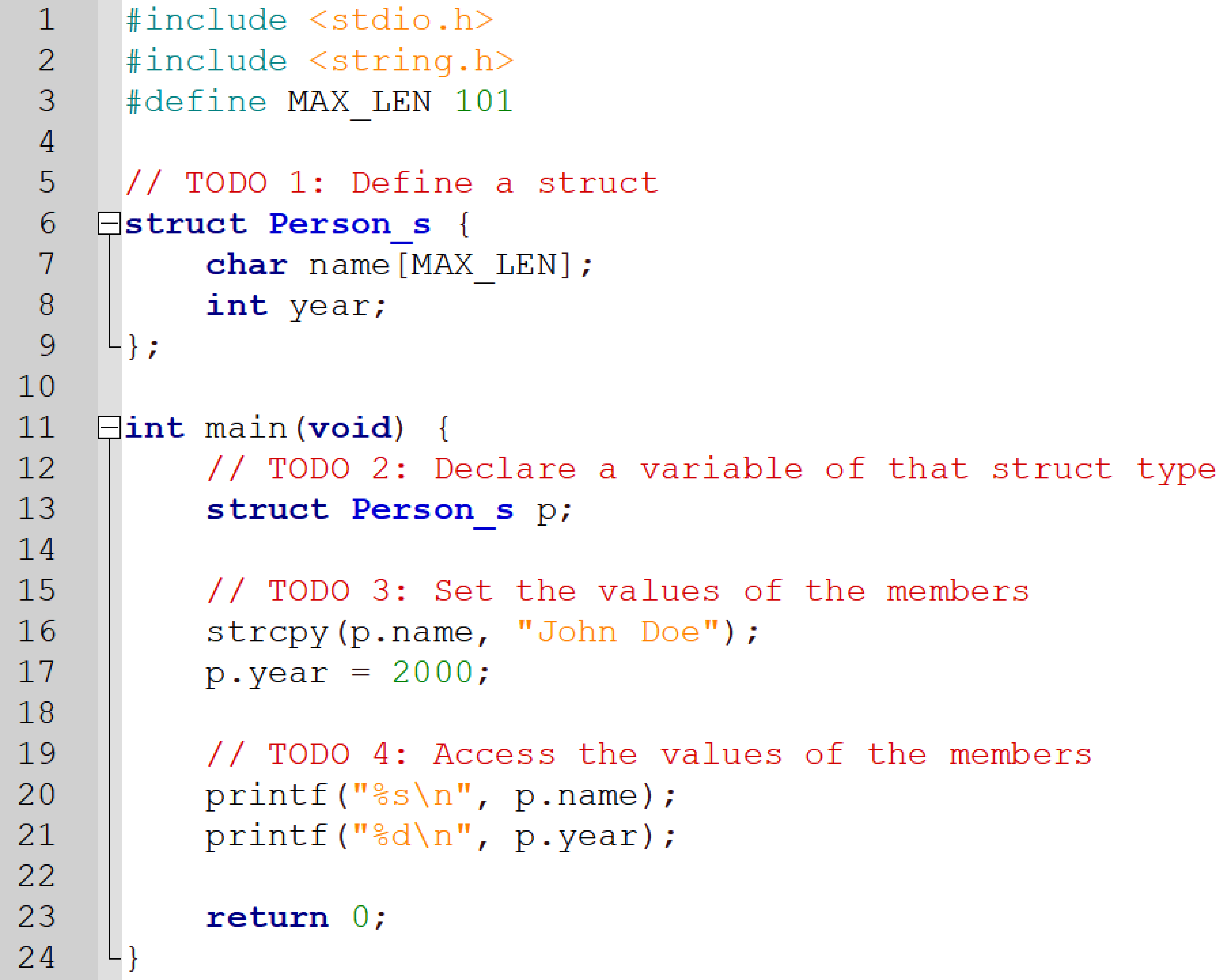
- Declare a struct **`Person_s`** variable

- Set the values for the members

The Dot Operator

- To access a member of a struct, we use the `.` (**dot operator**)

- Sometimes referred to as *member selection operator*



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

// TODO 1: Define a struct
struct Person_s {
    char name[MAX_LEN];
    int year;
};

int main(void) {
    // TODO 2: Declare a variable of that struct type
    struct Person_s p;

    // TODO 3: Set the values of the members
    strcpy(p.name, "John Doe");
    p.year = 2000;

    // TODO 4: Access the values of the members
    printf("%s\n", p.name);
    printf("%d\n", p.year);

    return 0;
}
```

Discussion

- Imagine declaring 10 **`Person_s`** variables

- We have been typing the **`struct`** keyword repeatedly

struct keyword is needed

```
struct Person_s p1;
struct Person_s p2;
struct Person_s p3;
        :
struct Person_s p10;
```

The `typedef` Keyword

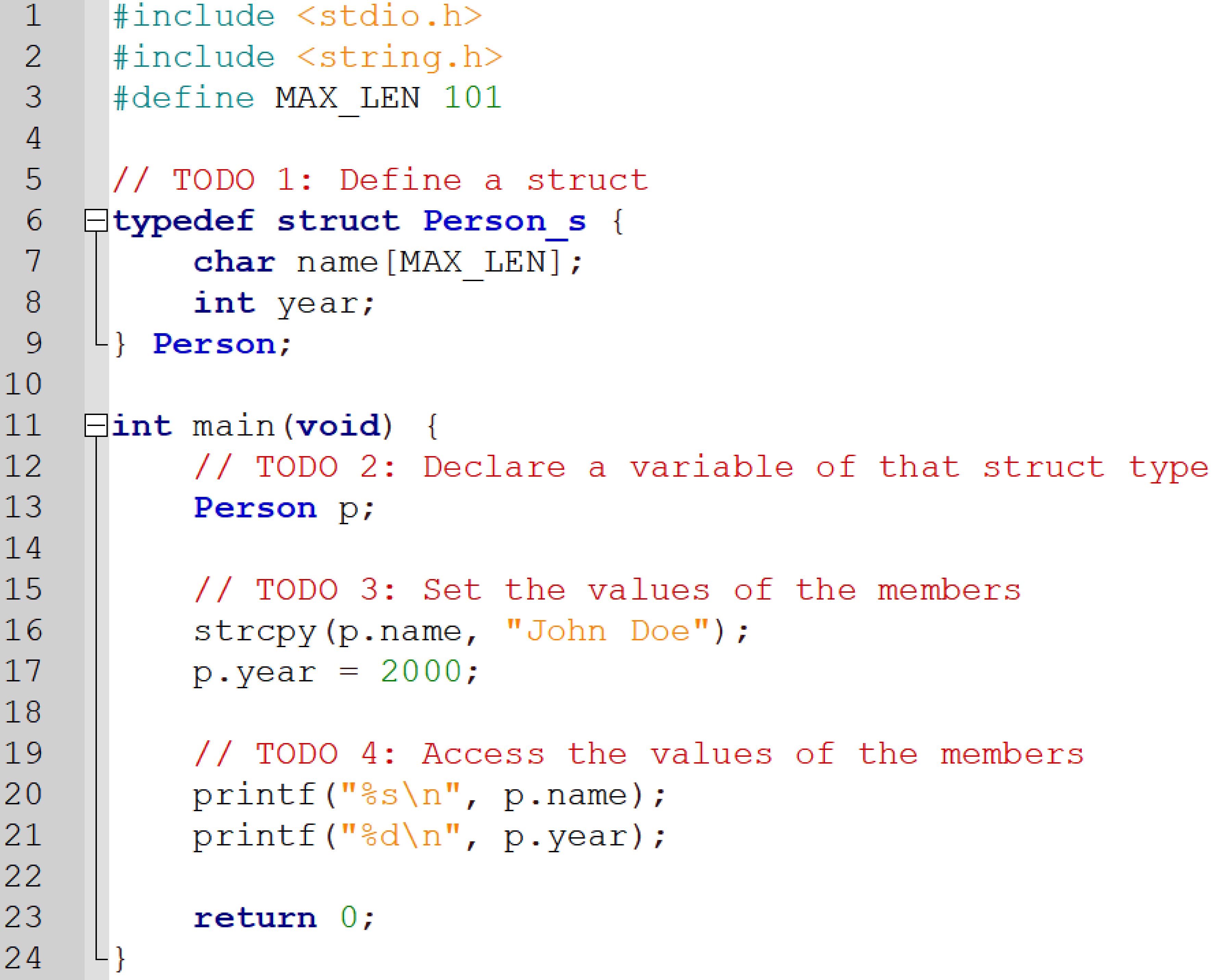
Creates an **alias** for an existing type

It is a declaration statement and does not create a new type

Syntax: typedef existing alias ;

Practice

Create a typedef for the **`struct Person_s`** so that the type can be referred to simply as **`Person`**.



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

// TODO 1: Define a struct
typedef struct Person_s {
    char name[MAX_LEN];
    int year;
} Person;

int main(void) {
    // TODO 2: Declare a variable of that struct type
    Person p;

    // TODO 3: Set the values of the members
    strcpy(p.name, "John Doe");
    p.year = 2000;

    // TODO 4: Access the values of the members
    printf("%s\n", p.name);
    printf("%d\n", p.year);

    return 0;
}
```

Practice /1

Define a function **`get_age(Person p)`** that returns the current age of person **`p`**. It should display the following:

Practice /2

Define a function **`introduce(Person p)`** that prints out the name and the current age of person **`p`**. It should display the following:

```
Hi, I'm [name]. I'm currently [age] years old.
```

Interlude

- Parallel arrays work but structures give us a more natural way to group related data

- Let’s take a moment to understand how arrays of structures work

Array of Structures

Just like an ordinary data type, you can create an array of structs

Visualization

Person people[5];

	.name	.year
people[0]	"John Doe"	2010
people[1]	"Jane Smith"	2005
people[2]	"Robert Smith"	1995
people[3]	?	?
people[4]	?	?

Practice /1

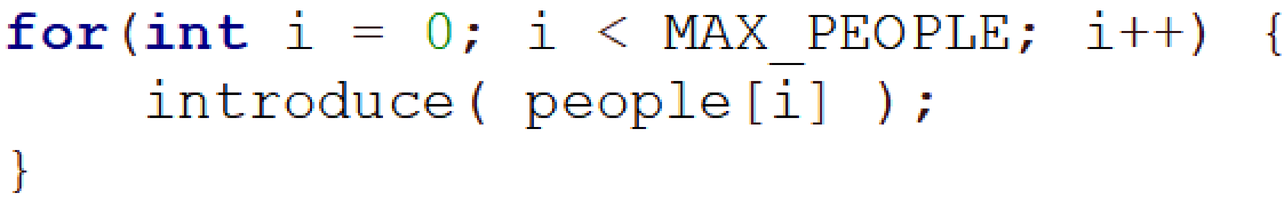
Solve the previous problem using array of structs

Put the solution first in the **`main()`** function

Afterward, define an appropriate function

Practice /2

Given that there is an **`introduce()`** function, traverse through all the persons and invoke this function.



```
for(int i = 0; i < MAX_PEOPLE; i++) {
    introduce( people[i] );
}
```

Discussion

- What if we want to include some **data validation**?

- For example, if the year is invalid, set it to a default value of 1900

- This leads to spilling over some logic on our **`main()`** function

- We want to separate this logic (recall: **modularization**)

Functions that Return a Structure

- You can write functions that returns a value whose data type is user-defined

- In our case, we can return a **`Person`**

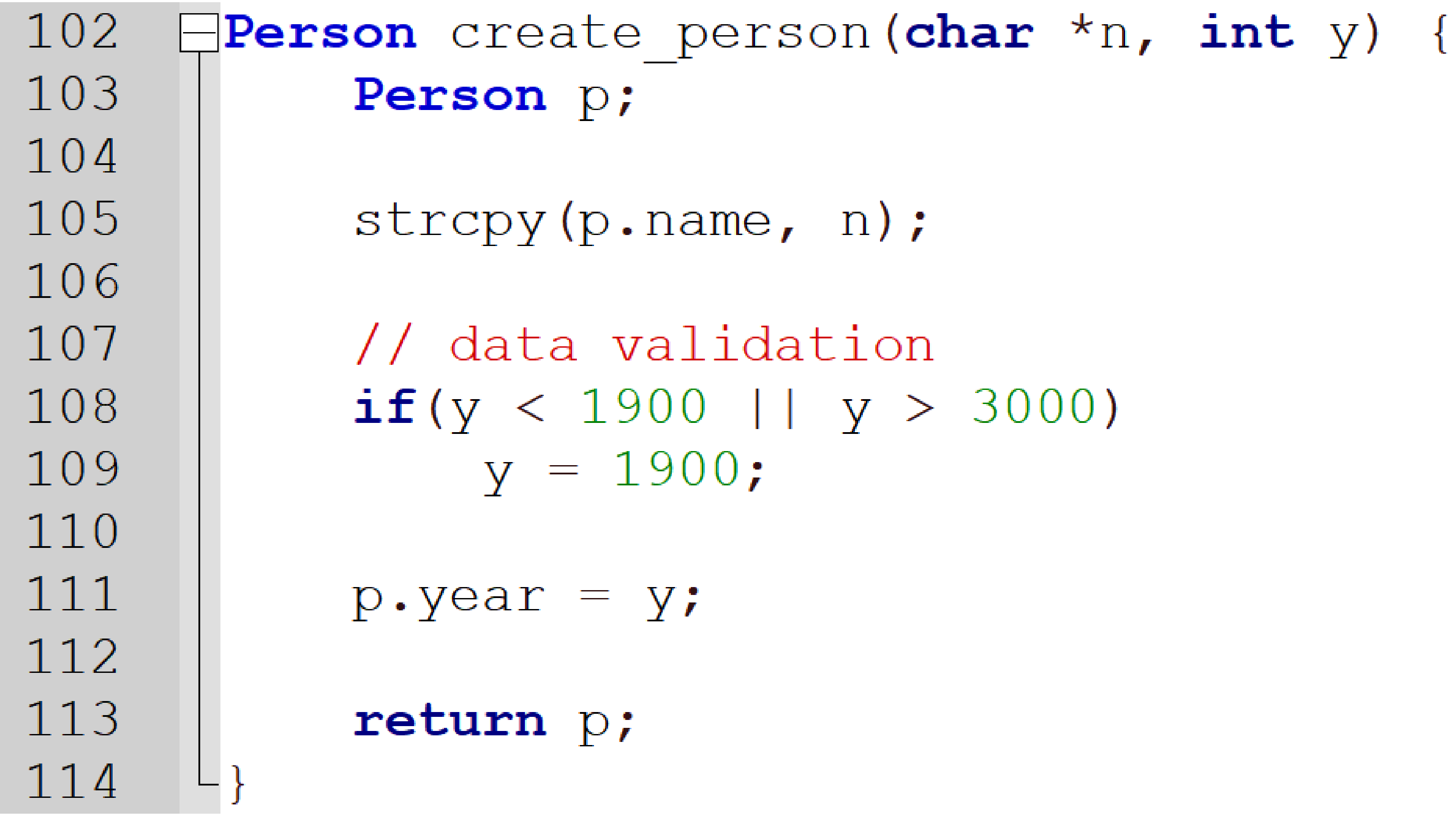
Practice

Define a function **`create_person()`** that takes two inputs: **`name`** and **`year`**. It returns a **`Person`** with these values. If the **`year`** is invalid, return a **`Person`** with birth year of **`1900`**.

```
Person create_person(char *name, int year) {



}
```



```c
Person create_person(char *n, int y) {
    Person p;

    strcpy(p.name, n);

    // data validation
    if(y < 1900 || y > 3000)
        y = 1900;

    p.year = y;

    return p;
}
```

Notes

- The **design pattern** illustrated separated the logic of the validation

- Notice how the calling function doesn’t really care about how validation works

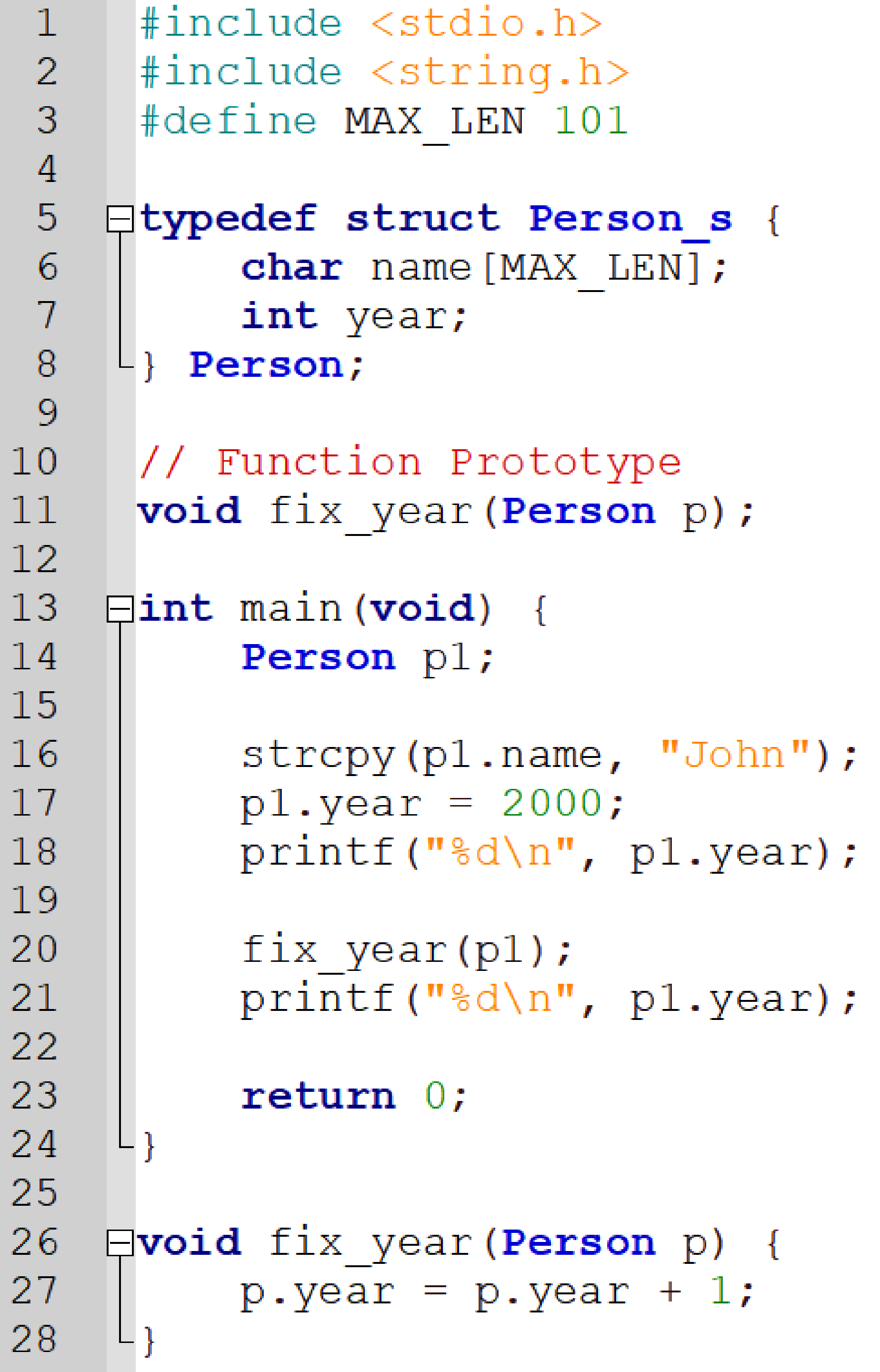
- It just needs to be able to work with an existing person

Discussion

What if we realized that there was an off-by one issue with the year information in the file?

Code Tracing

What is the output?



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

typedef struct Person_s {
    char name[MAX_LEN];
    int year;
} Person;

// Function Prototype
void fix_year(Person p);

int main(void) {
    Person p1;

    strcpy(p1.name, "John");
    p1.year = 2000;
    printf("%d\n", p1.year);

    fix_year(p1);
    printf("%d\n", p1.year);

    return 0;
}

void fix_year(Person p) {
    p.year = p.year + 1;
}
```

Notes /1

- Recall the concept of **pass-by-value**
- Any modifications done by the called function will not be reflected or seen by the calling function
- Just like ordinary variables, the function received a **copy** of the variable (i.e., it has its own copy with same values)

Notes /2

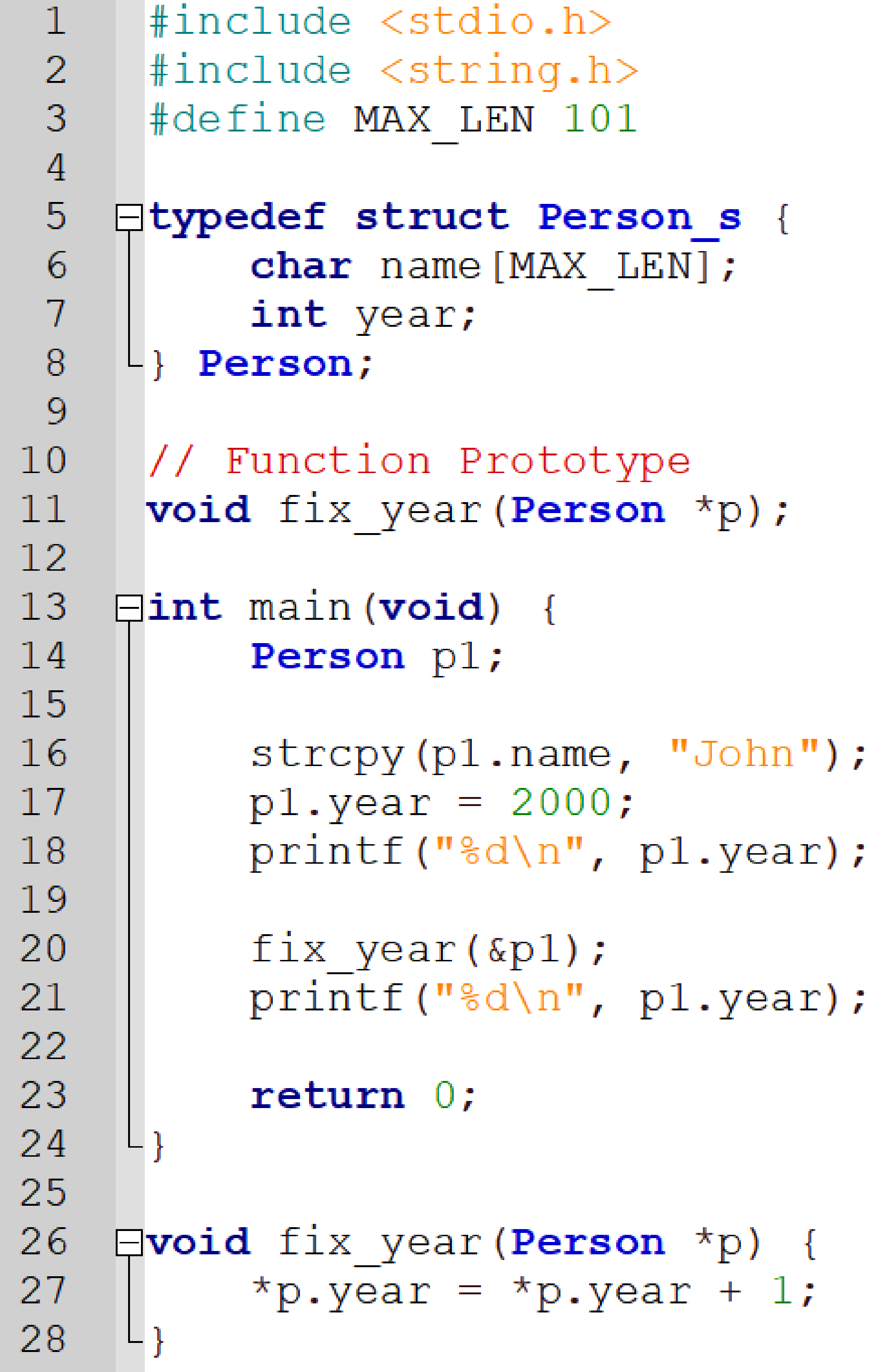
- Therefore, if you want to make the modifications seen by the calling function, you must do a **pass-by-reference**

- The same idea, you pass the address of the variable

Practice

- Update the code to allow for pass-by-reference

- Also, update **`main()`** so that it calls this function instead



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

typedef struct Person_s {
    char name[MAX_LEN];
    int year;
} Person;

// Function Prototype
void fix_year(Person *p);

int main(void) {
    Person p1;

    strcpy(p1.name, "John");
    p1.year = 2000;
    printf("%d\n", p1.year);

    fix_year(&p1);
    printf("%d\n", p1.year);

    return 0;
}

void fix_year(Person *p) {
    *p.year = *p.year + 1;
}
```

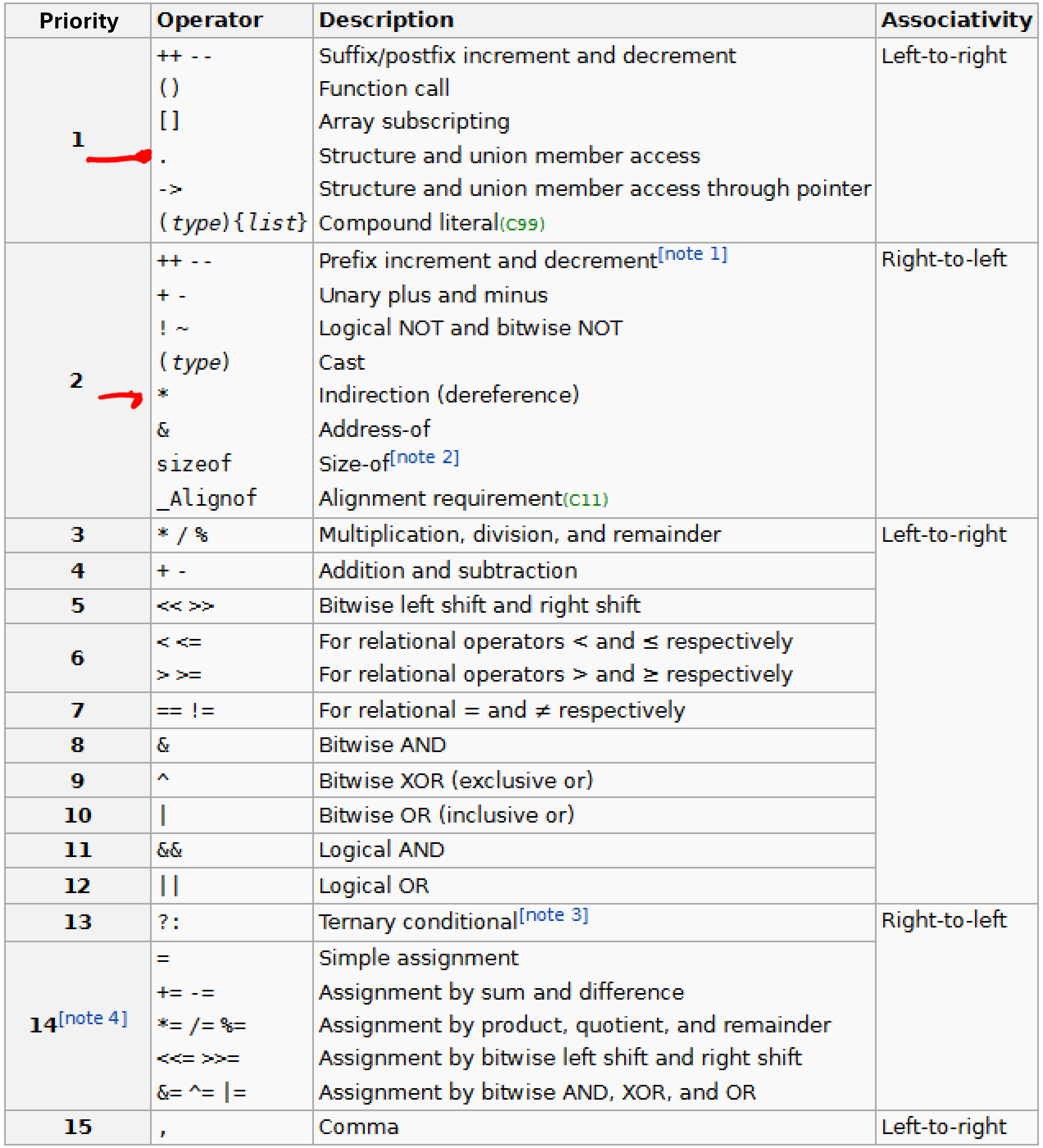
Notes

- We encountered a **syntax error**

- It has something to do with the **order of precedence** of operators

- The `.` (dot) has a higher precedence than **`*`** (dereference)

- How do we solve this?



Priority	Operator	Description	Associativity
1	++ -- () [] . -> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement[note 1] Unary plus and minus Logical NOT and bitwise NOT Cast Indirection (dereference) Address-of Size-of[note 2] Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Arrow Operator

A shortcut that does the same thing is the **->** (**arrow operator**)

Common Error

- Ensure you know when to use the dot and the arrow operators

- Also, this will be critical in an advanced course and when we discuss **dynamic memory allocation**

Discussion /1

- When designing solutions, it is often better to do a **pass by reference**

- In our previous example, if we are passing a structure that has 10 members (or fields) to a function, we are using 10 additional memory spaces

Discussion /2

- However, if we pass by reference, we are only passing the address, thereby, not using up a lot of memory

- Observe this by using the **`sizeof()`** operator to see the size of the variable received by the called function

Practice

Show two similar functions and illustrate the output of the **`sizeof`** operator

Refer to Webcourses for the C Code: Structures Experiment

Your Turn!

Solve the challenge posed earlier in this slide deck.

Scenario /1

- Say for example, now we want to keep track additional information about the person

- We want store the person’s complete date of birth

- How do we do this?

Scenario /2

- We can modify our person structure to add 2 new members
- Another approach is to create another structure related to dates
- So, we define a type date then add a new member to the person

people.txt

```
3
John Doe
2010 1 2
Jane Smith
2005 2 27
Robert Smith
1995 7 29
```

Hierarchical Structures

- A structure can also have a member that is also a structure

- At times, the order when you define the structures may matter

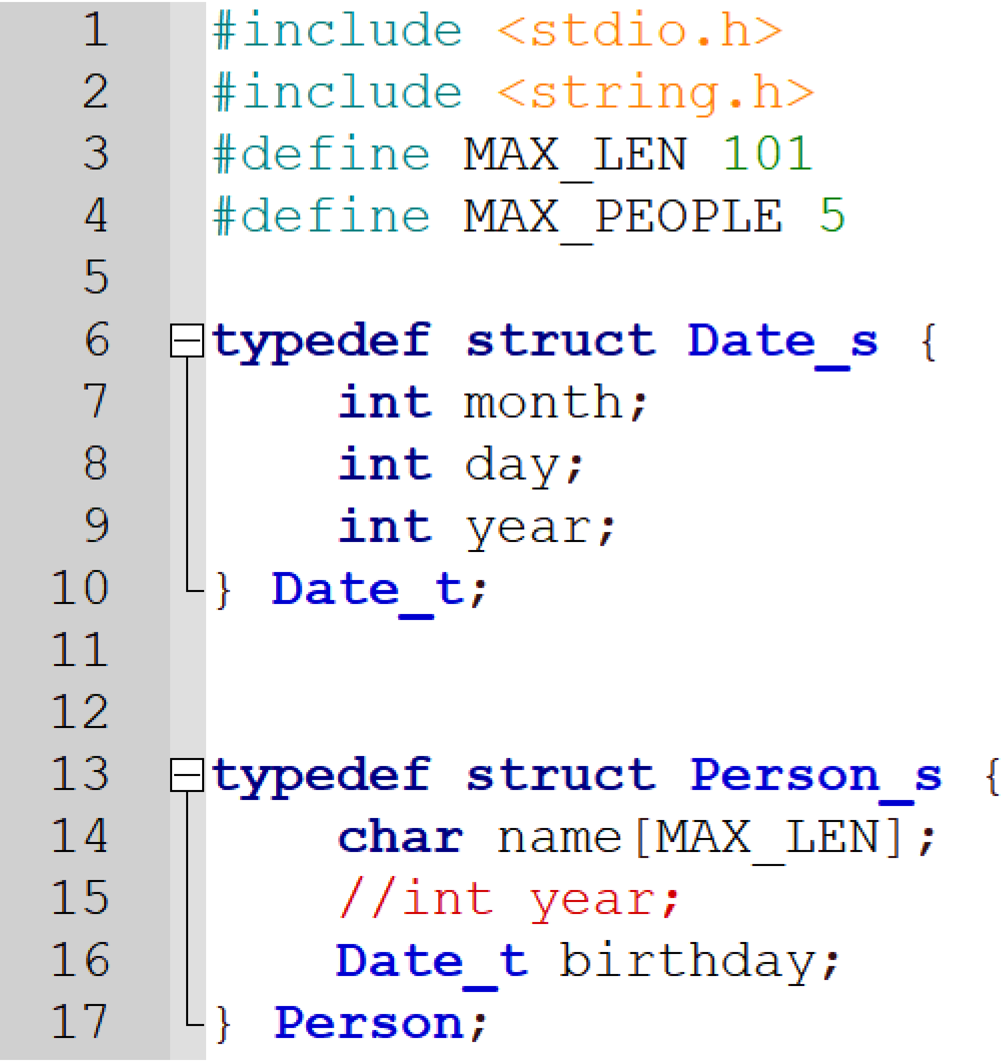
Practice

- Define an additional structure called **`Date_t`** with 3 members

- Update the **`Person`** structure so that it includes the **`Date_t`** type

- Modify the **`main()`** function to include the new information

- **Optional:** Remove the **`year`** member in the **`Person`**



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101
#define MAX_PEOPLE 5

typedef struct Date_s {
    int month;
    int day;
    int year;
} Date_t;


typedef struct Person_s {
    char name[MAX_LEN];
    //int year;
    Date_t birthday;
} Person;
```

Refer to Webcourses for the C Code: People Database (Structures)

Your Turn!

Define a function **`create_date()`** that takes three integers: **`year`**, **`month`**, and **`day`**. It returns a **`Date_t`** with these values. If the input is invalid, return a **`Date_t`** set to January 1, 1900.

```
Date_t create_date(int year, int month, int day) {



}
```

Questions?